State Machine Replication is More Expensive Than Consensus

Karolos Antoniadis
EPFL, Lausanne, Switzerland
karolos.antoniadis@epfl.ch

Rachid Guerraoui
EPFL, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Dahlia Malkhi
VMware Research, Palo Alto, USA
dmalkhi@vmware.com

Dragos-Adrian Seredinschi
EPFL, Lausanne, Switzerland
dragos-adrian.seredinschi@epfl.ch

Abstract

Consensus and State Machine Replication (SMR) are generally considered to be equivalent problems. In certain system models, indeed, the two problems are computationally equivalent: any solution to the former problem leads to a solution to the latter, and vice versa.

In this paper, we study the relation between consensus and SMR from a *complexity* perspective. We find that, surprisingly, completing an SMR command can be more expensive than solving a consensus instance. Specifically, given a synchronous system model where every instance of consensus always terminates in constant time, completing an SMR command does *not* necessarily terminate in constant time. This result naturally extends to partially synchronous models. Besides theoretical interest, our result also corresponds to practical phenomena we identify empirically. We experiment with two well-known SMR implementations (Multi-Paxos and Raft) and show that, indeed, SMR is more expensive than consensus in practice. One important implication of our result is that—even under synchrony conditions—no SMR algorithm can ensure bounded response times.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Consensus, State machine replication, Synchronous model

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.7

1 Introduction

Consensus is a fundamental problem in distributed computing. In this problem, a set of distributed processes need to reach agreement on a single value [32]. Solving consensus is one step away from implementing State Machine Replication (SMR) [31, 49]. Essentially, SMR consists of replicating a sequence of commands—often known as a log—on a set of processes which replicate the same state machine. These commands represent the ordered input to the state machine. SMR has been successfully deployed in applications ranging from storage systems, e.g., LogCabin built on Raft [43], to lock [13] and coordination [27] services. At a high level, SMR can be viewed as a sequence of consensus instances, so that each value output from an instance corresponds to a command in the SMR log.

From a *solvability* standpoint and assuming no malicious behavior, SMR can use consensus as a building block. When the latter is solvable, the former is solvable as well (the reverse direction is straightforward). Most previous work in this area, indeed, explain how to build SMR assuming a consensus foundation [21, 33, 36], or prove that consensus is equivalent from a solvability perspective with other SMR abstractions, such as atomic broadcast [14, 42]. An important body of work also studies the complexity of individual consensus instances [22, 28, 35, 47]. SMR is typically assumed to be a repetition of infinitely many consensus instances [29, 34] augmented with a reliable broadcast primitive [14], so at first glance it seems that the complexity of an SMR command can be derived from the complexity of the underlying consensus. We show that this is not the case.

In practice, SMR algorithms can exhibit irregular behavior, where some commands complete faster than others [12, 40, 54]. This suggests that the complexity of an SMR command can vary and may not necessarily coincide with the complexity of consensus. Motivated by this observation, we study the relation between consensus and SMR in terms of their *complexity*. To the best of our knowledge, we are the first to investigate this relation. In doing so, we take a formal, as well as a practical (i.e., experimental) approach. Counter-intuitively, we find that SMR is not necessarily a repetition of consensus instances.

We show that completing an SMR command can be more expensive than solving a consensus instance. Constructing a formalism to capture this result is not obvious. We prove our result by considering a fully synchronous system, where every consensus instance always completes in a *constant number of rounds*, and where at most one process in a round can be suspended (e.g., due to a crash or because of a network partition). A suspended process in a round is unable to send or deliver any messages in that round. Surprisingly, in this system model, we show that it is impossible to devise an SMR algorithm that can complete a command in constant time, i.e., completing a command can potentially require a *non-constant number of rounds*. We also discuss how this result applies in weaker models, e.g., partially synchronous, or if more than one process is suspended per round (see Section 3.2).

At a high level, the intuition behind our result is that a consensus instance "leaks," so that some processing for that instance is deferred for later. Simply put, even if a consensus instance terminates, some protocol messages belonging to that instance can remain undelivered. Indeed, consensus usually builds on majority quorum systems [51], where a majority of processes is sufficient and necessary to reach agreement; any process which is not in this majority may be left out. Typically, undelivered messages are destined to processes which are not in the active majority—e.g., because they are slower, or they are partitioned from the other processes. Such a leak is inherent to consensus: the instance must complete after gathering a majority, and should not wait for additional processes. If a process is not in the active majority, that process might as well be faulty, e.g., permanently crashed.

In the context of an SMR algorithm, when successive consensus instances leak, the same process can be left behind across multiple SMR commands; we call this process a *straggler*. Consequently, the deferred processing accumulates. It is possible, however, that this straggler is in fact correct. This means that eventually the straggler can become part of the active quorum for a command. This can happen when another process fails and the quorum must switch to include the straggler. When such a switch occurs, the SMR algorithm might not be able to proceed before the straggler recovers the whole chain of commands that it misses. Only after this recovery completes can the next consensus instance (and SMR command) start. Another way of looking at our result is that a consensus instance can neglect stragglers, whereas SMR must deal with the potential burden of helping stragglers catch-up.[1]

We experimentally validate our result in two well-known SMR systems: a Multi-Paxos implementation (LibPaxos [4]) and a Raft implementation (etcd [2]). Our experiments include the wide-area and clearly demonstrate the difference in complexity, both in terms of latency and number of messages, between a single consensus instance and an SMR command. Specifically, we show that even if a single straggler needs to be included in an active quorum, SMR performance noticeably degrades. It is not unlikely for processes to become stragglers in practical SMR deployments, since these algorithms typically run on commodity networks [7]. These systems are subject to network partitions, processes can be slow or crashed, and consensus-based implementations can often be plagued with corner-cases or implementation issues [9, 13, 25, 30], all of which can lead to stragglers.

Our contribution in this paper is twofold. First, we initiate the study of the relation, in terms of complexity, between consensus and SMR. We devise a formalism to capture the difference in complexity between these two problems, and use this formalism to prove that completing a single consensus instance is not equivalent to completing an SMR command in terms of their complexity (i.e., number of rounds). More precisely, we prove that it is impossible to design an SMR algorithm that can complete a command in constant time, even if consensus always completes in constant time. Second, we experimentally validate our theoretical result using two SMR systems in both a single-machine and a wide-area network.

Roadmap. The rest of this paper is organized as follows. We describe our system model in Section 2. In Section 3 we present our main result, namely that no SMR algorithm can complete every command in a constant number of rounds. Section 4 presents experiments to support our result. We describe the implications of our result in Section 5, including ways to circumvent it and a trade-off in SMR. Finally, Section 6 concludes the paper.

2 Model

This paper studies the relation in terms of complexity between consensus and State Machine Replication (SMR). In this section we formulate a system model that enables us to capture this relation, and also provide background notions on consensus and SMR.

We consider a synchronous model and assume a finite and fixed set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$, where $|\Pi| = n \geq 3$. Processes communicate by exchanging messages. Each message is taken from a finite set $M = \{m_1, \ldots\}$, where each message has a positive and a bounded size, which means that there exists a $B \in \mathbb{N}^+$ such that $\forall m \in M, 0 < |m| \leq B$.

A process is a state machine that can change its state as a consequence of delivering a message or performing some local computation. Each process has access to a read-only global clock, called *round number*, whose value increases by one on every round. In each round, every process p_i: (1) sends one message to every other process $p_j \neq p_i$ (in total p_i sends $n-1$ messages in each round);[2] (2) delivers any messages sent to p_i in that round; and (3) performs some local computation.

An *algorithm* in such a model is the state machine for each process and its initial state. A *configuration* corresponds to the internal state of all processes, as well as the current round number. An *initial configuration* is a configuration where all processes are

[1] We note that this leaking property seems not only inherent in consensus, but in any equivalent replication primitive, such as atomic broadcast.

[2] As a side note, if a process p_i does not have something to send to process p_j in a given round, we simply assume that p_i sends an empty message.

in their initial state and the round number is one. In each round, up to $n(n-1)$ messages are transmitted. More specifically, we denote a transmission as a triplet (p, q, m) where $p, q \in \Pi (p \neq q)$ and $m \in M$. For instance, transmission $(p_i, p_j, m_{i,j})$ captures the sending of message $m_{i,j}$ from process p_i to process p_j. We associate with each round an *event*, corresponding to the set of transmissions which take place in that round; we denote this event by $\tau \subseteq \{(p_i, p_j, m_{i,j}) : i, j \in \{1, \dots, n\} \wedge i \neq j\}$. An *execution* corresponds to an alternating sequence of configurations and events, starting with an initial configuration. An execution e^+ is called an *extension* of a finite execution e if e is a prefix of e^+. Given a finite execution e, we denote with $E(e)$ the set of all extensions of e. We assume deterministic algorithms: the sequence of events uniquely defines an execution.

Failures. Our goal is to capture the *complexity*—i.e., cost in terms of number of synchronous rounds—of a consensus instance and of an SMR command, and expose any differences in terms of this complexity. Towards this goal, we introduce a failure mode which omits *all* transmissions to and from at most one process per round.

We say that a process p_i is *suspended in round* r associated with the event τ, if $\forall m \in M$ and $\forall j \in \{1, \dots, n\}$ with $j \neq i$, $(p_i, p_j, m) \notin \tau$ and $(p_j, p_i, m) \notin \tau$, hence $|\tau| = n(n-1) - 2(n-1) = (n-1)(n-2)$. If a process p_i is not suspended in a round r, we say that p_i is *correct in round* r. In a round associated with an event τ where all processes are correct there are no omissions, hence $|\tau| = n(n-1)$. A process p_i is *correct* in a finite execution e if there is a round in e where p_i is correct. Process p_i is *correct* in an infinite execution e if there are infinitely many rounds in e where p_i is correct. For our result, it suffices that in each round a single process is suspended. Note that each round in our model is a communication-closed layer [18], so messages omitted in a round are not delivered in any later round.

A suspended process represents a scenario where a process is slowed down. This may be caused by various real-world conditions, e.g., a transient network disconnect, a load imbalance, or temporary slowdown due to garbage collection. In all of these, after a short period, connections are dropped and message buffers are reclaimed; such conditions can manifest as message omissions. The notion of being suspended also represents a model where processes may crash and recover, where any in-transit messages are typically lost.

There is a multitude of work [3, 44, 45, 47, 48] on message omissions (e.g., due to link failures) in synchronous models. Our system model is based on the mobile faults model [44]. Note however that our model is stronger than the mobile faults model, since we consider that either exactly zero or exactly $2(n-1)$ message omissions occur in a given round.[3] Other powerful frameworks, such as layered analysis [41], the heard-of model [15], or RRFD [20] can be used to capture omission failures, but we opted for a simpler approach that can specifically express the model which we consider.

2.1 Consensus

In the consensus problem, processes have initial values which they propose, and have to decide on a single value. Consensus [10] is defined by three properties: validity, agreement, and termination. Validity requires that a decided value was proposed by one of the processes, whilst agreement asks that no two processes decide differently. Finally, termination states that every correct process eventually decides. In the interest of having an "apples to apples" comparison with SMR commands (defined below, Section 2.2), we introduce a client (e.g., learner in Paxos terminology [33]), and say that a consensus instance completes as soon

[3] If a process p is suspended, then $n-1$ messages sent by p and $n-1$ messages delivered to p are omitted.

as the client learns about the decided value. This client is not subject to being suspended, and after receiving the decided value, the client broadcasts this value to the other processes. Algorithm 1 is a consensus algorithm based on this idea.

It is easy to see that in such a model consensus completes in two rounds: processes broadcast their input, and every process uses some deterministic function (e.g., maximum) to decide on a specific value among the set of values it delivers. Since all processes deliver exactly the same set of $n-1$ (or n) values, they reach agreement. In the second round, all processes send their decided value (a process that was suspended in the first round might send $\perp$) to all the other processes, including the client. Since $n \geq 3$ and at least $n-1$ processes are correct in the second round, the client delivers the decided value (i.e., a value that is not $\perp$) and thus the consensus instance completes by the end of round two. Afterwards (starting from the third round), the client broadcasts the decided value to all the processes, so eventually every correct process decides, satisfying termination. Note that if a process is suspended in the first round (but correct in the second round), it will decide in the second round, after delivering the decided value from some other process. Algorithm 1 represents this solution in which the red and blue lines correspond to the synchronous model's send and deliver actions respectively.

We remark that Algorithm 1 does not contradict the lossy link impossibility result of Santoro and Widmayer [44], even though our model permits more than $n-1$ message omissions in a round, since the model we consider is stronger.

Algorithm 1 Consensus

```
procedure PROPOSE($p_i, v_i$)  ▷ $p_i$ proposes value $v_i$
    $decision \leftarrow \perp$
    ▷ round 1
    $\forall p \in \Pi \setminus \{p_i\}$, send$(p, v_i)$  ▷ $\Pi$ is the set of processes
    $values \leftarrow \{v_i\} \cup \{$ each value $v$ delivered from process $p$ $(\forall p \in \Pi \setminus \{p_i\})$ $\}$
    if $|values| \neq 1$ then  ▷ $p_i$ is correct in round 1
        $decision \leftarrow deterministicFunction(values)$
    else  ▷ $p_i$ was suspended
        ▷ $p_i$ cannot decide yet
    ▷ round $k$ $(k \geq 2)$: consensus instance completes in round 2
    $\forall p \in (\Pi \setminus \{p_i\}) \cup \{client\}$, send$(p, decision)$  ▷ broadcast decided value
    $values \leftarrow \{decision\} \cup \{$ each decision $d$ delivered from process $p$ $(\forall p \in \Pi \setminus \{p_i\})$ $\}$
    $decision \leftarrow d$ where $d \in values$ and $d \neq \perp$
```

We emphasize that although correct processes can decide in the first round, we consider that the consensus instance *completes* when the client delivers the decided value. Hence, the consensus instance in Algorithm 1 completes in the second round. In more practical terms, this consensus instance has a constant cost.

2.2 State Machine Replication

The SMR approach requires a set of processes (i.e., replicas) to agree on an ordered sequence of commands [31, 49]. We use the terms replica and process interchangeably. Informally, each replica has a log of the commands it has performed, or is about to perform, on its copy of the state machine.

Log. Each replica is associated with a sequence of decided and known commands which we call the *log*. The commands are taken from a finite set $C = \{c_1, \dots, c_k\}$. We denote the log with $\ell(e,p)$ where e is a finite execution, p is a replica, and each element in $\ell(e,p)$ belongs to the set $C \cup \{\epsilon\}$. Specifically, $\ell(e,p)$ corresponds to commands known by replica p after all the events in a finite execution e have taken place (e.g., $\ell(e,p) = c_{i_1}, \epsilon, c_{i_3}$). For $1 \leq i \leq \left|\ell(e,p)\right|$, we denote with $\ell(e,p)_i$ the i-th element of sequence $\ell(e,p)$. If there is an execution e and $\exists p \in \Pi$ and $\exists i \in \mathbb{N}^+$ such that $\ell(e,p)_i = \epsilon$, this means that replica p does not have knowledge of the command for the i-th position in its log, while at least one replica does have knowledge of this command (i.e., $\exists p' \neq p \in \Pi : \ell(e,p')_i \neq \epsilon$). We assume that if a process knows about a command c, then c exists in $\ell(e,p)$. To keep our model at a high-level, we abstract over the details of how each command appears in the log of each replica, since this is typically algorithm-specific. Additionally, state-transfer optimizations or snapshotting [43] are orthogonal to our discussion.

An SMR algorithm is considered *valid* if the following property is satisfied for any finite execution e of that algorithm: $\forall p, p' \in \Pi$ and for every i such that $1 \leq i \leq min(\left|\ell(e,p)\right|, \left|\ell(e,p')\right|)$, if $\ell(e,p)_i \neq \ell(e,p')_i$ then either $\ell(e,p)_i = \epsilon$ or $\ell(e,p')_i = \epsilon$. In other words, consider a replica p which knows a command for a specific log position i, i.e., $\ell(e,p)_i = c_k$, where $c_k \in C$. Then for the same log position i, any other process p' can either know command c_k (i.e., $\ell(e,p')_i = c_k$), not know the command (i.e., $\ell(e,p')_i = \epsilon$), or have no information regarding the command (i.e., $\left|\ell(e,p')\right| < i$). In this paper, we only consider valid SMR algorithms.

In what follows, we define what it means for a replica to be a *straggler*, as well as how replicas first learn about commands.

Stragglers. Intuitively, stragglers are replicas that are missing commands from their log. More specifically, let L be $\max_p |\ell(e,p)|$. We say that q is a *k-straggler* if the number of non-ϵ elements in $\ell(e,q)$ is at most $L-k$. A replica p is a *straggler* in an execution e if there exists a $k \geq 1$ such that p is a k-straggler. Otherwise, we say that the replica is a *non-straggler*. A replica that is suspended for a number of rounds could potentially miss commands and hence become a straggler.

Client. Similar to the consensus client, there is a client process in SMR as well. In SMR, however, the client proposes commands. The client acts like the $(n+1)$-th replica in a system with n replicas and its purpose is to supply one command to the SMR algorithm, wait until it receives (i.e., delivers) a response for the command it sent, then send another command, etc. A client, however, is different from the other replicas, since an SMR algorithm has no control over the state machine operating in the client and the client is never suspended. A client operates in lock-step[4] as follows:

- sends a command $c \in C$ to all the n replicas in some round r;
- waits until some replica responds to the client's command (i.e., the response of applying the command).[5]

A replica p can respond to a client command c only if it has all commands preceding c in its log. This means that $\exists i : \ell(e,p)_i = c$ and $\forall j < i, \ell(e,p)_j \neq \epsilon$. We say that the client is *suggesting* a command c at a round r if the client sends a message containing command c to all the replicas in round r. Similarly, we say that a client *gets a response* for command

[4] Clients need not necessarily operate in lock-step, but can employ pipelining, i.e., can have multiple commands outstanding. Practical systems employ pipelining [2, 4, 43], and we account for this aspect later in our practical experiments of Section 4.

[5] We consider that a command is applied instantaneously on the state machine (i.e., execution time for any command is zero).

c at a round r if some replica sends a message to the client containing the response of the command c in round r.

SMR Algorithm. Algorithm 1 shows that consensus is solvable in our model. It seems intuitive that SMR is solvable in our model as well. To prove that this is the case, we introduce an SMR algorithm. Roughly speaking, this algorithm operates as follows. Each replica contains an ordered log of decided commands. A command is decided for a specific log position by executing a consensus instance similar to Algorithm 1. The SMR algorithm takes care of stragglers through the use of helping. Specifically, each replica tries to help stragglers by sending commands which the straggler might be missing. Due to space constraints, we defer the detailed description and the proof of the SMR algorithm, which can be seen as a contribution in itself, to our corresponding technical report [5]. As we show next (Section 3), no SMR algorithm can respond to a client in a finite number of rounds. Hence, even with helping, our SMR algorithm cannot guarantee a constant response either. Finally, note that our definition of a valid SMR algorithm does not include a liveness property since this is not needed for our result. Nevertheless, the SMR algorithm we propose guarantees that if a client suggests a command, then the client eventually gets a response.

3 Complexity Lower Bound on State Machine Replication

We now present the main result of our paper. Roughly speaking, we show that there is no State Machine Replication (SMR) algorithm that can always respond to a client in a constant number of rounds. We also discuss how this result extends beyond the model of Section 2.

3.1 Complexity Lower Bound

We briefly describe the idea behind our result. We observe that there is a bounded number of commands that can be delivered by a replica in a single round, since messages are of bounded size, a practical assumption (Lemma 1). Using this observation, we show that in a finite execution e, if each replica p_i is missing β_i commands, then an SMR algorithm needs $\Omega(\min_i \beta_i)$ rounds to respond to at least one client command suggested in an extension $e^+ \in E(e)$ (Lemma 2). Finally, for any $r \in \mathbb{N}^+$, we show how to construct an execution e where each replica misses enough commands in e, so that a command suggested by a client in an extension $e^+ \in E(e)$ cannot get a response in less than r rounds (Theorem 3). Hence, no SMR algorithm in our model can respond to every client command in a constant number of rounds.

▶ **Lemma 1.** *A single replica can deliver up to a bounded number (that we denote by Ψ) of commands in a round.*

Proof. Since any message m is of bounded size B ($\forall m \in M, |m| \leq B$), the number of commands message m can contain is bounded. Let us denote with ψ the maximum number of commands any message can contain. Since the number of commands that can be contained in one message is at most ψ, a replica can transmit at most ψ commands to another replica in one round. Therefore, in a given round a replica can deliver from other replicas up to $\Psi = (n-1)\psi$ commands. In other words, a replica cannot recover faster than Ψ commands per round. ◀

▶ **Lemma 2.** *For any finite execution e, if each replica p_i is a β_i-straggler (i.e., p_i misses β_i commands), then there is a command suggested by the client in some execution $e^+ \in E(e)$ such that we need at least $\lceil \min_i(\beta_i/\Psi) \rceil$ rounds to respond to it.*

Proof. Consider an execution $e^+ \in E(e)$ such that in a given round r, a client suggests to all replicas a command c, where round r exists in e^+ but does not exist in e. This implies that replicas are not yet aware of command c in e, so command c should appear in a log position i where i is greater than $\max_p |\ell(e,p)|$. In order for a replica to respond to the client's command c, the replica first needs to have all the commands preceding c in its log. For this to happen, some replica needs to get informed about β_i commands. Note that from Lemma 1, a replica can only deliver Ψ commands in a round. Therefore, a replica needs at least $\lceil \beta_i/\Psi \rceil$ rounds to get informed about the commands it is missing (i.e., recover), and hence we need at least $\lceil \min_i(\beta_i/\Psi) \rceil$ rounds for the client to get a response for c. ◀

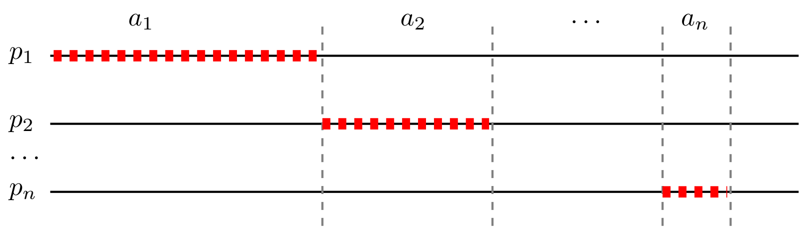


Figure 1 Constructed execution of Theorem 3. Red dashed lines correspond to rounds where a replica is suspended. Replica p_1 is suspended for a_1 rounds, replica p_2 for a_2 rounds, etc.

▶ **Theorem 3.** *For any $r \in \mathbb{N}^+$ and any SMR algorithm with n replicas ($n \geq 3$), there exists an execution e, such that a command c which the client suggests in some execution $e^+ \in E(e)$ cannot get a response in less than r rounds.*

Proof. Assume by contradiction that, given an SMR algorithm, each command suggested by a client needs at most a constant number of rounds k to get a response. Since we can get a response to a command in at most k rounds, we can make a replica "miss" any number of commands by simply suspending it for an adequate amount of rounds.

To better convey the proof we introduce the notion of a phase. A phase is a conceptual construct that corresponds to a number of contiguous rounds in which a specific replica is suspended. Specifically, we construct an execution e consisting of n phases. Figure 1 conveys the intuition behind this execution. In the i-th phase, replica p_i is suspended for α_i rounds, and $\alpha_i \neq \alpha_j$ for $i \neq j$. The idea is that after the n-th phase, each replica is a straggler and needs more than k rounds to become a non-straggler and be able to respond to a client command suggested in a round o, where o exists in e^+ but not in e. We start from the n-th phase, going backwards. In the n-th phase, we make replica p_n miss enough commands, say β_n. In general, the number β_n of commands is such, that if a client suggests a command at the end of the n-th phase, the client cannot get a response from within k rounds of the command being suggested. For this to hold, it suffices to miss $\beta_n = k\Psi + 1$ commands. In order to miss β_n commands, we have to suspend p_n for at least $\beta_n k$ rounds, since a client may submit a new command every (at most) k rounds. Thus, we set $\alpha_n = \beta_n k$. Similarly, replica p_{n-1} has to miss enough commands (β_{n-1}) such that it cannot get all the commands in less than k rounds. Note that after p_{n-1} was suspended for α_{n-1} rounds, replica p_n took part in α_n rounds. During these α_n rounds, replica p_{n-1} could have recovered commands it was missing. Therefore, p_{n-1} must miss at least $\beta_{n-1} = (\alpha_n + k)\Psi + 1$ commands and $\alpha_{n-1} = \beta_{n-1}k$. In the same vein, $\forall i \in \{1,\ldots,n\}$ $\beta_i = ((\sum_{j=i+1}^{n} \alpha_j) + k)\Psi + 1$.

With our construction we succeed in having $\beta_i/\Psi = (\sum_{j=i+1}^{n} \alpha_j) + k + 1/\Psi > k$ for every $i \in \{1, \ldots, n\}$. Therefore, using Lemma 2, after the n phases, each replica needs more than k rounds to get informed about commands it is missing from its log, a contradiction. ◀

Theorem 3 states that there exists no SMR algorithm in our model that can respond to every client command in a constant number of rounds.

3.2 Extension to other Models

The system model we use in this paper (Section 2) lends itself to capture naturally the difference in complexity (i.e., number of rounds) between consensus and SMR. It is natural to ask whether this difference extends to other system models—and which are those models. Identifying all the models where our result applies, or does not apply, is a very interesting topic which is beyond the scope of this paper, but we briefly discuss it here.

Consider models which are stronger than ours. An example of a stronger model is one that is synchronous with no failures; such a model would disallow stragglers and hence both consensus and SMR can be solved in constant time. Similarly, if the model does not restrict the size of messages (see Lemma 1), then an SMR command can complete in constant time, circumventing our result. We further discuss how our result can be circumvented in Section 5.

A more important case is that of weaker, perhaps more realistic models. If the system model is too weak—if consensus is not solvable [19]—then it is not obvious how consensus relates to SMR in terms of complexity. Such a weak model, however, can be augmented, for instance with unreliable failure detectors [14], allowing consensus to be solved. Informally, during well-behaved executions of such models, i.e., executions when the system behaves synchronously and no failures occur [28], SMR commands can complete in constant time.

Most practical SMR systems [13, 16, 40, 43] typically assume a partially synchronous or an asynchronous model with failure detectors [14], and executions are not well-behaved, because failures are prone to occur [7]. We believe our result applies in these practical settings, concretely within synchronous periods (or when the failure detector is accurate, respectively) of these models. During such periods, if at least one replica can suffer message omissions, completing an SMR command can take a non-constant amount of time. Indeed, in the next section, we present an experimental evaluation showing that our result holds in a partially synchronous system.

4 The Empirical Perspective

Our goal in this section is to substantiate empirically the theoretical result of Section 3. We first cover details of the experimental methodology. Then we discuss the evaluation results both in a single-machine environment, as well as on a practical wide-area network (WAN).

4.1 Experimental Methodology

We use two well-known State Machine Replication (SMR) systems: (1) LibPaxos, a Multi-Paxos implementation [4], and (2) etcd [2], a mature implementation of the Raft protocol [43]. We note that LibPaxos distinguishes between three roles of a process: proposer, acceptor, and learner [33]. To simplify our presentation, we unify the terminology so that we use the term *replica* instead of *acceptor*, the term *client* replaces *learner*, and the term *leader* replaces *proposer*. Each system we deploy consists of three replicas, since this is sufficient to validate our result and moreover it is a common deployment practice [16, 23]. We employ one client. In LibPaxos, we use a single leader, which corresponds to a separate role from replicas. In Raft, one of the three replicas acts as the leader.

Using these two systems, we measure how consensus relates to SMR in terms of cost in the following three scenarios:

1. **Graceful**: when network conditions are uniform and no failures occur; this scenario only applies to the single-machine experiments of Section 4.2;
2. **Straggler**: a single replica is slower than the others (i.e., this is a straggler) but no failures occur, so the SMR algorithm needs not rely on the straggler;
3. **Switch**: a single replica is a straggler and a failure occurs, so the SMR algorithm has to include the straggler on the critical path of agreement on commands.

Due to the difficulty of running synchronous rounds in a practical system, our measurements are not in terms of rounds (as in the model of Section 2). Instead, we take a lower-level perspective. We report on the *cost*, i.e., number of messages, and the *latency* measured at the client.[6] Specifically, in each experiment, we report on the following three measurements.

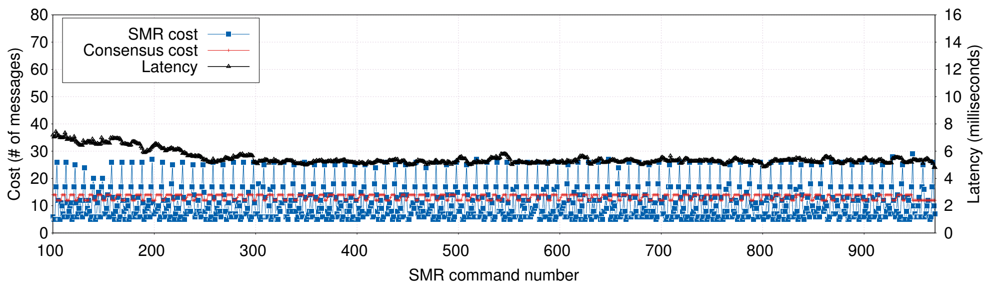
First, we present the cost of each consensus instance i in terms of number of messages which belong to instance i, and which were exchanged between replicas, as well as the client. Each consensus instance has an identifier (called *iid* in LibPaxos and *index* in Raft), and we count these messages up to the point where the instance completes at the client. Recall that in our model (Section 2.1) we similarly consider consensus to complete when the client learns the decided value. This helps us provide an “apples to apples” comparison between the cost of consensus instances and SMR commands (which we describe next).

Second, we measure the cost of each SMR command c. Each command c is associated with a consensus instance i. The cost of c is similar to the cost of i: we count messages exchanged between replicas and the client for instance i.[7] The cost of a command c, however, is a more nuanced measurement. As we discussed already, a consensus instance typically leaks messages, which can be processed later. Also, both systems we consider use pipelining, so that a consensus instance i may overlap with other instances while a replica is working on command c. Specifically, the cost of c can include messages leaked from some instance j, where $j < i$ (because a replica cannot complete command c without having finished all previous instances) but also from some instance k, with $k > i$ (these future instances are being prepared in advance in a pipeline, and are not necessary for completing command c).

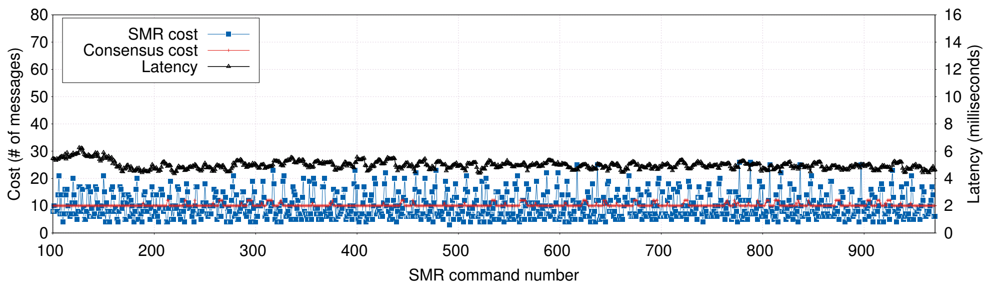
Third, we measure the latency for completing each SMR command. An SMR command starts when the client submits this command to the leader, and ends when the client learns the command. In LibPaxos, this happens when the client gathers replies for that command from two out of three replicas; in Raft, the leader notifies the client with a response.

We consider both a single-machine setup and a WAN. The former setup serves as a controlled environment where we can vary specifically the variable we seek to study, namely the impact of a straggler when quorums switch. For this experiment, we use LibPaxos and we discuss the results thoroughly. The latter setup reflects real-world conditions which we use to validate against our findings in the single-machine setup, and we experiment with both systems. In all executions the client submits 1000 SMR commands; we ignore the first 100 (warm-up) and the last 50 commands (cool-down) from the results. We run the same experiment three times to confirm that we are not presenting outlying results.

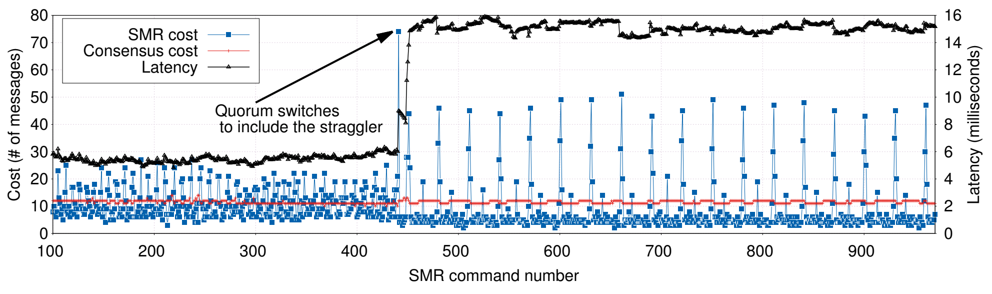
[6] Note that it is simple to convert rounds to messages, considering our description of rounds in Section 2.
[7] For LibPaxos, the cost of consensus and SMR includes additionally messages involving the leader.



(a) Graceful scenario: all replicas experience uniform conditions and no failures occur.



(b) Straggler scenario: one of the three replicas is a straggler.



(c) Switch scenario: one of the three replicas is a straggler and the active quorum switches to include this straggler.

Figure 2 Experimental results with LibPaxos on a single-machine setup. We compare the cost of SMR commands with the cost of consensus instances in three scenarios.

4.2 Experimental Results on a Single Machine

We experiment on an Intel Core i7-3770K (3.50GHz) equipped with 16GB of RAM. Since there is no network in these experiments, spurious network conditions—which can arise in practice, as we shall see next in Section 4.3—do not create noise in our results. To make one of the replicas a straggler, we make this replica relatively slower through a random delay (via the `select` system call) of up to $500us$ when this replica processes a protocol message.

In Figure 2a we show the evolution of the three measurements we study for the **graceful** execution. The mean latency is $5590us$ with a standard deviation of $730us$, i.e., the performance is very stable. This execution serves as a baseline.

In Figure 2b we present the result for the **straggler** scenario. The average latency, compared with Figure 2a, is slightly smaller, at $5005us$; the standard deviation is $403us$. The explanation for this decrease is that there is less contention (because the straggler backs-off periodically), so the performance increases. In this scenario, additionally, there is more variability in the cost of SMR commands, which is a result of the straggler replica being less predictable in how many protocol messages it handles per unit of time.

For both Figures 2a and 2b, the average cost of an SMR command is the same as the average cost of a consensus instance, specifically around 12 messages. There is, however, a greater variability in the cost of SMR commands—ranging from 5 to 30 messages—while consensus instances are more regular—between 11 and 13 messages. As we mentioned already, the variability in the cost of SMR springs from two sources: (1) consensus instances leak into each other, and (2) the use of pipelining, a crucial part in any practical SMR algorithm, which allows consensus instances to overlap in time [27, 46].

Pipelining allows the leader to have multiple outstanding proposals, and these are typically sent and delivered in a burst, in a single network-level packet. This means that some commands can comprise just a few messages (all the other messages for such a command have been processed earlier with previous commands, or have been deferred), whereas some commands comprise many more messages (e.g., messages leaked from previous commands, or processed in advance from upcoming commands). In our case, the pipeline has size 10, and we can distinguish in the plots that the bumps in the SMR cost have this frequency. Larger pipelines allow higher variability in the cost of SMR. Importantly, to reduce the effect of pipelining on the cost of SMR commands, this pipeline size of 10 is much smaller than it is used in practice, which can be 64, 128, or larger [2, 4].

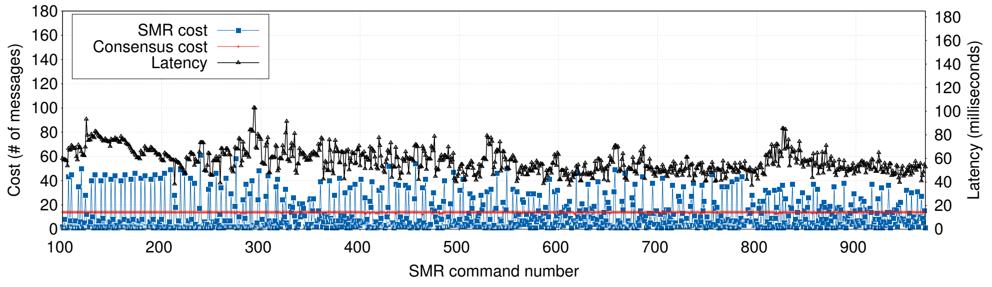
Figure 2c shows the execution where we stop one replica, so the straggler has to take part in the active quorum. The moment when the straggler has to recover all the missing state and start participating is evident in the plot. This happens at SMR command 450. We observe that SMR command 451 has considerably higher cost. This cost comprises all the messages which the straggler requires to catch-up, before being able to participate in the next consensus instance. The cost of consensus instance 451 itself is no different than other consensus instances. Since the straggler becomes the bottleneck, the latency increases and remains elevated for the rest of the execution. The average latency in this case is noticeably higher than in the two previous executions, at $10730us$ (standard deviation of $4726us$). For this execution, we observe the same periodical bumps in the cost of SMR commands. Because the straggler replica is on the critical path of agreement, these bumps are more pronounced and less frequent: the messages concerning the straggler (including to and from other replicas or the client) accumulate in the incoming and outgoing queues and are processed in bursts.

4.3 Wide-area Experiments

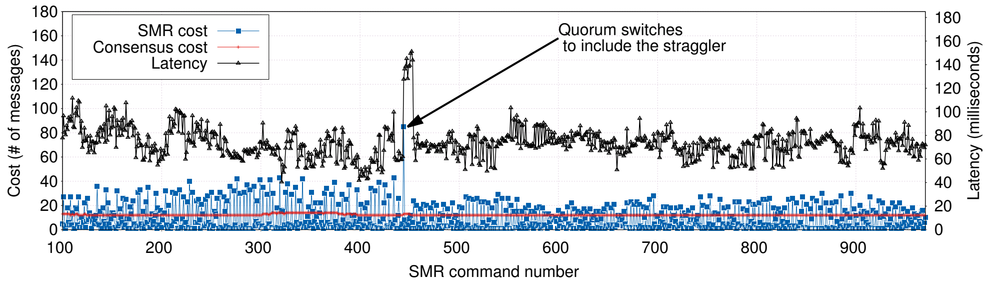
We deploy both LibPaxos and Raft on Amazon EC2 using *t2.micro* virtual machines [1]. For LibPaxos, we colocate the leader with the client in Ireland, and we place the three replicas in London, Paris, and Frankfurt, respectively. Similarly, for Raft we colocate the leader replica along with the client in Ireland, and we place the other two replicas in London and Frankfurt. Under these deployment conditions, the replica in Frankfurt is naturally the straggler, since this is the farthest node from Ireland (where the leader is in both systems). Therefore, we do not impose any delays, as we did in the earlier single-machine experiments. Furthermore, colocating the client with the leader minimizes the latency between these two, so the latency measurements we report indicate the actual latency of SMR.

Figures 3 and 4 present our results for LibPaxos and Raft, respectively. To enhance visibility, please note that we use different scales for the *y* and *y2* axes. These experiments do not include the **graceful** scenario, because the WAN is inherently heterogeneous.

The most interesting observation is for the **switch** scenarios, i.e., Figures 3b and 4b. In these experiments, when we stop one of the replicas at command 450, there is a clear spike in the cost of SMR, which is similar to the spike in Figure 2c. Additionally, however, there is also a spike in latency. This latency spike does not manifest in single-machine experiments,



(a) Straggler scenario: the replica in Frankfurt is a straggler, since this is the farthest from the leader in Ireland. The system forms a quorum using the replicas in London and Paris.



(b) Switch scenario: at SMR command 450 we switch out the replica in London. The straggler in Frankfurt then becomes part of the active quorum.

Figure 3 Experimental results with LibPaxos on the WAN. Similar to Figure 2, we compare the cost of SMR commands with the cost of consensus instances.

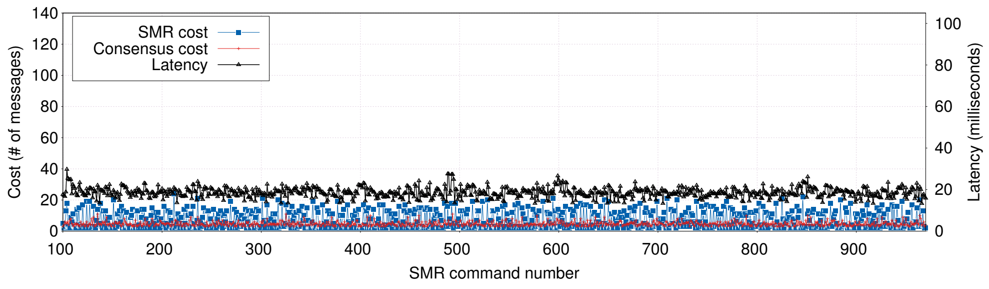
where communication delays are negligible. Moreover, on the WAN the latency spike extends over multiple commands, because the system has a pipeline so the latency of each command being processed in the pipeline is affected while the straggler is catching up. After this spike, the latency decreases but remains slightly more elevated than prior to the switch, because the active quorum now includes the replica from Frankfurt, which is slightly farther away; the difference in latency is roughly $5ms$.

Beside the latency spike at SMR command 450, these experiments reveal a few other glitches, for instance around command 830 in Figure 3a, or command 900 in Figure 4b. In fact, we observe that unlike our single-machine experiments, the latency exhibits a greater variability. As we mentioned already, this has been observed before [12, 40, 54] and is largely due to the heterogeneity in the network and the spurious behavior this incurs. This effect is more notable in LibPaxos, but Raft also shows some variability. The latter system reports consistently lower latencies because an SMR command completes after a single round-trip between the leader and replicas [43].

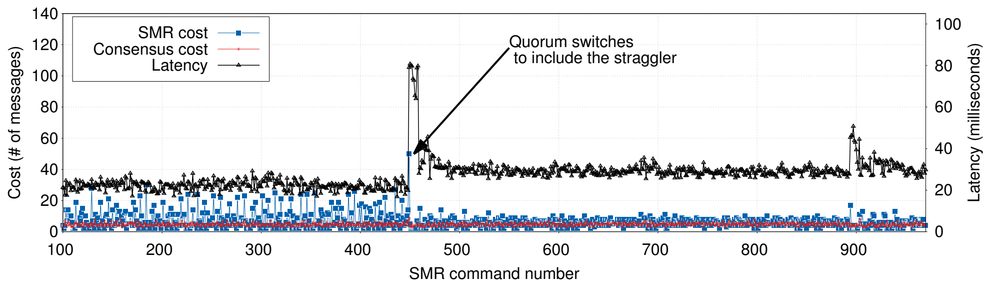
As a final remark, our choice of parameters is conservative, e.g., execution length or pipeline width. For instance, in executions longer than 1000 commands we can exacerbate the difference in cost between SMR commands and consensus instances. Longer executions allow a straggler to miss even more state which it needs to recover when switching.

5 Discussion

The main implication of Theorem 3 is that it is impossible to devise a State Machine Replication (SMR) algorithm that can bound its response times. There are several conditions, however, which allow to circumvent our lower bound, which we discuss here. Moreover,



(a) Straggler scenario: the replica in Frankfurt is a straggler. The active quorum consists of the leader in Ireland and the replica in London.



(b) Switch scenario: we stop the replica in London at SMR command 450. Thereafter, the active quorum must switch to include the straggler in Frankfurt.

Figure 4 Experimental results with Raft on the WAN. Similar to Figures 2 and 3, we compare the cost of SMR commands with the cost of consensus instances.

when our result does apply, we observe that SMR algorithms can mitigate, to some degree, the performance degradation in the worst-case, i.e., when quorums switch and stragglers become necessary. These algorithms experience a trade-off between best-case and worst-case performance. We also discuss how various SMR algorithms deal with this trade-off.

Circumventing the Lower Bound. Informally, our result applies to SMR systems which fulfill two basic characteristics: i) messages are bounded in size, and ii) replicas can straggle for arbitrary lengths of time. Simply put, if one of these conditions does not hold, then we can circumvent Theorem 3. We discuss several cases when this can happen.[8]

For instance, if the total size of the state machine is bounded, as well as small in size, then the whole state machine can potentially fit in a single message, so a straggler can recover in bounded time. This is applicable in limited practical situations. We are not aware of any SMR protocol that caps its state. But this state can be very small in some applications, e.g., if SMR is employed towards replicating only a critical part of the application, such as distributed locks or coordination kernels [27, 39].

The techniques of load shedding or backpressure [53] can be employed to circumvent our result. These are application-specific techniques which, concretely, allow a system to simply drop or deny a client command if the system cannot fulfill that command within bounded time. Other, more drastic, approaches to enforce strict latencies involve resorting to weak consistency or combining multiple consistency models in the same application [24],

[8] We do not argue that we can guarantee bounded response times in a general setting, only in the model we consider in Section 2.

or provisioning additional replicas proactively when stragglers manifest [17, 50].

Best-case Versus Worst-case Performance Trade-off. When our lower bound holds, an SMR algorithm can take steps to ameliorate the impact which stragglers have on performance in the worst-case (i.e., when quorums switch). Coping with stragglers, however, does not come for free. The best-case performance can suffer if this algorithm expends resources (e.g., additional messages) to assist stragglers. Concretely, these resources could have been used to sustain a higher best-case throughput. When a straggler becomes necessary in an active quorum, however, this algorithm will suffer a smaller penalty for switching quorums and hence the performance in the worst-case will be more predictable.

This is the trade-off between best- and worst-case performance, which can inform the design of SMR algorithms. Most of the current well-known SMR protocols aim to achieve superior best-case throughput by sacrificing worst-case performance. This is done by reducing the replication factor, also known as a *thrifty* optimization [40]. In this optimization, the SMR system uses only $F + 1$ instead of $2F + 1$ replicas—thereby stragglers are non-existent—so as to reduce the amount of transmitted messages and hence improve throughput or other metrics [4, 38, 40]. In the worst-case, however, when a fault occurs, this optimization requires the SMR system to either reconfigure or provision an additional replica on the spot [37, 38], impairing performance.

Multi-Paxos proposes a mode of operation that can strike a good balance between best- and worst-case performance [32]. Namely, replicas in this algorithm can have gaps in their logs. When gaps are allowed, a replica can participate in the agreement for some command on log position k even if this replica does not have earlier commands, i.e., commands in log positions l with $l < k$. As long as the leader has the full log, the system can progress. Even when quorums switch, stragglers can participate without recovery. If the leader fails, however, the protocol halts [11, 52] because no replica has the full log, and execution can only resume after some replica builds the full log by coordinating with the others. It would be interesting in future work to experiment with an implementation that allows gaps, but LibPaxos does not follow this approach [4], and we are not aware of any such implementation.

It is interesting to note that there is not much work on optimizing SMR performance for the worst-case, e.g., by expediting recovery [11], and this is a good avenue for future research, perhaps with applicability in performance-sensitive applications. We believe SMR algorithms are possible where replicas balance among themselves the burden of keeping each other up to date collaboratively, e.g., as attempted in [8]. This would minimize the amount of missing state overall (and at any single replica), so as to be prepared for the worst-case, while minimizing the impact on the best-case performance.

6 Concluding Remarks

We examined the relation between consensus and State Machine Replication (SMR) in terms of their complexity. We proved the surprising result that SMR is more expensive than a repetition of consensus instances. Concretely, we showed that in a synchronous system where a single instance of consensus always terminates in a constant number of rounds, completing one SMR command can potentially require a non-constant number of rounds. Such a scenario can occur if some processes are stragglers in the SMR algorithm, but later the stragglers become active and are necessary to complete a command. We showed that such a scenario can occur if even one process is a straggler at a time.

Our result—that an SMR algorithm cannot guarantee a constant response time, even if otherwise the system behaves synchronously—brought into focus a trade-off in SMR.

In a nutshell, this is the trade-off between the best-case performance and the worst-case performance of an SMR algorithm. On the one hand, such an algorithm can optimize for the worst-case performance. In this case, the algorithm can dedicate resources (e.g., by provisioning additional processes or assisting stragglers) to preserve its performance even when faults manifest, translating into lower tail latencies; there are certain classes of SMR-based applications where latencies and their variability are very important [6, 16, 17]. On the other hand, an SMR algorithm can optimize for best-case performance, i.e., during fault-free periods, so that the algorithm advances despite stragglers being left arbitrarily behind [26, 40]. This strategy means that the algorithm can achieve superior throughput, but its performance will be more sensible to faults.

Additionally, we supported our formal proof with experimental results using two well-known SMR implementations (a Multi-Paxos and a Raft implementation). Our experiments highlighted the difference in cost between a single consensus instance and an SMR command. To the best of our knowledge, we are the first to formally—as well as empirically—investigate the performance-cost difference between consensus and SMR.

Acknowledgments. We wish to thank our colleagues from the Distributed Computing Laboratory (DCL) for the many fruitful discussions. We are also thankful towards Willy Zwaenepoel and Jad Hamza, as well as the anonymous reviewers for providing us with helpful comments.

References

1 Amazon EC2. http://aws.amazon.com/ec2/. [Online; accessed 9-May-2018].

2 etcd. https://github.com/coreos/etcd. [Online; accessed 9-May-2018].

3 Gracefully degrading consensus and k-set agreement in directed dynamic networks. *Theoretical Computer Science*, 726.

4 LibPaxos3. https://bitbucket.org/sciascid/libpaxos. [Online; accessed 9-May-2018].

5 Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State Machine Replication is More Expensive than Consensus. Technical Report, EPFL, 2018. https://github.com/agms18/tr/raw/master/report.pdf.

6 B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *DSN*, 2017.

7 Peter Bailis and Kyle Kingsbury. The network is reliable. *ACM Queue*, 12(7):20, 2014.

8 Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *ATC*, 2013.

9 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *DSN*, 2014.

10 Christian Cachin, Rachid Guerraoui, and Luìs Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2011.

11 Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated agreement protocols for higher availability. In *Network Computing and Applications*, 2008.

12 Daniel Cason, Parisa J Marandi, Luiz E Buzato, and Fernando Pedone. Chasing the tail of atomic broadcast protocols. In *SRDS*, 2015.

13 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.

14 Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

15 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, Apr 2009.

16 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3), 2013.

17 Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

18 Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2(3):155 – 173, 1982.

19 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

20 Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *PODC*, 1998.

21 Álvaro García-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. Paxos consensus, deconstructed and abstracted (extended version). *CoRR*, abs/1802.05969, 2018.

22 Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the complexity of asynchronous gossip. In *PODC*, 2008.

23 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP*, 2003.

24 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *OSDI*, 2016.

25 Heidi Howard and Jon Crowcroft. Coracle: Evaluating Consensus at the Internet Edge. In *SIGCOMM*, 2015.

26 Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *OPODIS*, 2016.

27 Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

28 Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: Preliminary version. *SIGACT News*, 32(2):45–63, June 2001.

29 M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

30 Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *EuroSys*, 2013.

31 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

32 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

33 Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

34 Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23. Springer, 2003.

35 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

36 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable Paxos. *TechReport, Microsoft Research*, 2008.

37 Leslie Lamport and Mike Massa. Cheap paxos. In *DSN*, 2004.

38 Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *SOSP*, 1991.

39 John MacCormick, Nick Murphy, Marc Najork, Chandu Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

40 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *SOSP*, 2013.

41 Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM Journal on Computing*, 31(4):989–1021, 2002.

42 A. Mostefaoui and M. Raynal. Low cost consensus-based atomic broadcast. In *Proceedings. 2000 Pacific Rim International Symposium on Dependable Computing*, 2000.

43 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *ATC*, 2014.

44 Nicola Santoro and Peter Widmayer. Time is not a healer. In *STACS*, 1989.

45 Nicola Santoro and Peter Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theoretical Computer Science*, 384(2):232 – 249, 2007.

46 Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. In *International Conference on Distributed Computing and Networking*, pages 153–167. Springer, 2012.

47 Ulrich Schmid, Bettina Weiss, and Idit Keidar. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing*, 38(5):1912–1951, 2009.

48 Ulrich Schmid, Bettina Weiss, and John Rushby. Formally verified byzantine agreement in presence of link faults. In *ICDCS*, 2002.

49 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

50 Emil Sit, Andreas Haeberlen, Frank Dabek, Byung-Gon Chun, Hakim Weatherspoon, Robert Morris, M Frans Kaashoek, and John Kubiatowicz. Proactive Replication for Data Durability. In *IPTPS*, 2006.

51 Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

52 Gustavo M. D. Vieira, Islene C. Garcia, and Luiz Eduardo Buzato. Seamless paxos coordinators. *CoRR*, abs/1710.07845, 2017.

53 Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

54 Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, 2009.

Appendix

7 State Machine Replication Algorithm

We present an SMR algorithm for our model (Section 2). In contrast to consensus (Algorithm 1), the presented SMR algorithm is quite more involved. For clarity, the algorithm is presented in two parts: Algorithm 2 and Algorithm 3. In Algorithm 2, we present the local variables of each replica and the code each replica executes in every round, and in Algorithm 3 we present the two main procedures of the algorithm: PREPAREMESSAGES and ONRECEIVE.

The high-level overview of the algorithm is that processes decide on a command similar to Algorithm 1 and can help each other by sending commands to processes that are missing them. A bit more specific, each process contains an ordered log of decided commands. For a command to appear in the i-th position of this log, processes need to agree by performing consensus instance i. Processes propose a command for the next consensus instance they are missing and if this position is already decided, other processes will try to help them by sending them their missing commands. In order to be able to help, each process has information[9] on the next consensus instance each other process tries to decide upon.

Algorithm 2 State Machine Replication: Local Variables for Process p_i and Flow in Each Round

```
▷ Local Variables
$ins \leftarrow 1$   ▷ next consensus instance number to get a decision for
$maxIns \leftarrow 0$   ▷ greatest consensus instance number where a value is decided upon
$nextMissingIns[p] \leftarrow 1 (\forall p \in \Pi \setminus \{p_i\})$   ▷ next instance each process needs
$cmdsSet \leftarrow \emptyset$   ▷ set of commands that are received from the client
$cmdsDecided[i] \leftarrow \perp \; (\forall i \in \mathbb{N}^+)$   ▷ $cmdsDecided[i]$ is the command decided for consensus
    instance $i$, $\perp$ means no decision is known yet
$messageFor[p] \leftarrow \perp \; (\forall p \in (\Pi \setminus \{p_i\}) \cup \{client\})$   ▷ messages to be sent in each round
$SM \leftarrow initSM$   ▷ initialized state machine to be replicated
$myProposal \leftarrow (p_i, \perp, 0)$   ▷ $p_i$'s last proposal in the format of $(pid, value, instance)$
$clientResponses[i] \leftarrow \perp \; (\forall i \in \mathbb{N}^+)$   ▷ stores all the responses destined for the client
$lastResponse \leftarrow 1$   ▷ index of $clientResponses$

while new round do
    SEND($messageFor$)
    $responses \leftarrow$ RECEIVE()
    ONRECEIVE($responses, myProposal$)
    $(messageFor, myProposal) \leftarrow$ PREPAREMESSAGES()
```

We continue by describing the local variables (Algorithm 2). Variable *cmdsDecided* corresponds to a log of commands[10] and contains the commands (in order) the process knows have been decided. Note that *cmdsDecided* allows gaps, for example, a process might have $cmdsDecided[1] \neq \perp$ and $cmdsDecided[3] \neq \perp$ but $cmdsDecided[2] = \perp$. It is guaranteed however that $\forall i \in \mathbb{N}^+ < ins, cmdsDecided[i] \neq \perp$, where *ins* corresponds to the smallest

[9] Note that since this information is local, it might become stale and not accurately describe the system.
[10] You can think of the log as the $\ell(e, p)$ construct presented in Section 2.

position in the log the process does not have a command. Similar to *ins*, *maxIns* corresponds to the maximum decided instance number of all the other processes. Specifically, *maxIns* contains the maximum number j such that a process has decided on a command for the j-th position in its log (i.e,. there is some process that has $cmdsDecided[j] \neq \perp$). Initially *maxIns* is zero, since no process has decided on a command. Each time a process sends a message, it attaches *maxIns* in it, so processes can get informed on the greatest consensus instance number for the whole system where a value has been decided upon. Additionally, each process attaches *ins* to each message it sends so that it can potentially get help from other processes. Helping takes place when a process informs other processes about decided commands they may need. For this, each process has an array (*nextMissing*) of the next instance each process needs. A process looks at this array and sees if it can help another process, and if so it sends the decided command to the other process. Variable *cmdsSet* corresponds to a set of commands that are received from the client and are to be proposed by the process. *messageFor* is set in PREPAREMESSAGES as we will see later on, and it simply contains the message to be sent in every round to the other processes or the client. Additionally, each replica has a state machine *SM* that is initialized to *initSM* and it provides an *apply* operation that takes as a parameter a command and returns the response of applying this command to the state machine. Variable *myProposal* corresponds to a potential command proposed by the process. Finally, the array *clientResponses* is used to store computed responses that are to be send to the client and *lastResponse* is used to index this array. This array is convenient in case a process is suspended in a round. If this is the case, a process cannot send the response to the client in this round, so the process keeps responses in the *clientResponses* array in order to be able to send a response when it is not suspended.

The exact steps an algorithm executes in a round are presented in lines 13-17: initially the process sends messages, then waits to receive back *responses* that are used together with *myProposal* to change the state of the process and compute the next round's messages (*messageFor*).

We continue by describing how PREPAREMESSAGES and ONRECEIVE operate. PREPAREMESSAGES operates as follows (Algorithm 3). First, it checks (Line 19) whether it has already decided for instance *ins*. This could happen, if the processes retrieved a decision in Line 60 or in Line 63 of ONRECEIVE. If the command exists in the set, the command is removed from it (lines 20-21). Afterwards, the command is applied to the state machine (Line 22), the response is stored in *clientResponses* (Line 23) and the latest response that has not yet been transmitted to the client is stored in *messageFor* (Line 24) so it can be sent in the next round to the client. Additionally, *ins* is incremented by one (Line 25). The algorithm then initializes *messageFor* to contain the pair $(ins, maxIns)$ (Line 26) for every message to be sent to each other process. Including this pair in each message is helpful, since *ins* allows other processes to know the consensus instance the process needs a command for, and *maxIns* ensures that processes only accept proposals for positions that have not yet been decided. Then, *myProposal* (Line 27) is cleared, since otherwise the algorithm might use a previous proposal message. Afterwards, if there exists a command (Line 28) to be proposed and that is not decided yet by the process (Line 30), the process concatenates the proposal to each message (the construct $\|$ corresponds to concatenation of messages) (Line 31), and sets *myProposal* (Line 32). Then, in lines 35 to 37, the process checks if it can help other processes by sending it decided commands it knows that other processes are missing. At the end it returns *messageFor* together with the proposal (Line 38). Finally, note that each message sent to another processes consists of at most three parts, a pair

Algorithm 3 State Machine Replication (for process p_i)

```
18: procedure PREPAREMESSAGES()
19:     if $cmdsDecided[ins] \neq \perp$ then
20:         if $cmdsDecided[ins] \in cmdsSet$ then
21:             $cmdsSet.remove(cmdsDecided[ins])$
22:         $response \leftarrow SM.apply(cmdsDecided[ins])$
23:         $clientResponses[ins] \leftarrow response$
24:         $messageFor[client] \leftarrow clientResponses[lastResponse]$
25:         $ins \leftarrow ins + 1$
26:     $messageFor[p] \leftarrow (ins, maxIns), \forall p \in \Pi \setminus \{p_i\}$
27:     $myProposal \leftarrow (p_i, \perp, 0)$  ▷ clear last proposal
28:     if $cmdsSet \neq \emptyset$ then
29:         $cmd \leftarrow cmdsSet.get()$  ▷ returns but does not remove an element from the set
30:         if $\forall j : cmdsDecided[j] \neq cmd$ then  ▷ not decided yet in which order to execute
    $cmd$
31:             $\forall p \in \Pi \setminus \{p_i\}$, $messageFor[p] \leftarrow messageFor[p] \parallel$ pro($cmd$, $ins$)
32:             $myProposal \leftarrow (p_i, cmd, ins)$
33:         else
34:             $cmdsSet.remove(cmd)$  ▷ already have decided on $cmd$, so no need to propose
    it
35:     for $p \in \Pi \setminus \{p_i\}$ do
36:         if $\exists j : nextMissingIns[p] = j \wedge cmdsDecided[j] = cmd \neq \perp$ then
37:             $messageFor[p] \leftarrow messageFor[p] \parallel$ dec($cmd$, $nextMissingIns[p]$)
38:     return $(messageFor, myProposal)$
39:
40: procedure ONRECEIVE($responses, myProposal$)
41:     if $(client, cmd) \in respones$ then
42:         $cmdsSet.put(cmd)$
43:         $responses \leftarrow responses \setminus \{(client, cmd)\}$
44:     ▷ a received message sent by process $p_j$ is of the format $p_j, A \| B \| C$, where
45:     ▷ $A = (ins_j, maxIns_j)$, $B =$pro$(cmd_j, ins_j)$ and $C =$dec$(cmd_j, nextMissing_j)$
46:     if $responses \neq \emptyset$ then  ▷ else, $p_i$ is suspended in this round
47:         if $messageFor[client] \neq \perp$ then
48:             $lastResponse \leftarrow lastResponse + 1$
49:             $messageFor[client] \leftarrow \perp$
50:         $proposals \leftarrow decisions \leftarrow \emptyset$
51:         for $p_j, (ins_j, maxIns_j) \|$pro$(pcmd_j, pins_j) \|$dec$(dcmd_j, nextMiss_j)$ in $responses$
    do
52:             $maxIns \leftarrow max(maxIns, maxIns_j)$
53:             $nextMissing[p_j] \leftarrow ins_j + 1$
54:             $proposals \leftarrow proposals \cup \{(p_j, pcmd_j, pins_j)\}$
55:             $decisions \leftarrow decisions \cup \{(dcmd_j, nextMiss_j)\}$
56:         $proposals \leftarrow proposals \cup \{myProposal\}$
57:         if $\exists(\_, \_, pins) \in proposals : pins = maxIns + 1$ then
58:             $commands \leftarrow \{pcmd : \exists(\_, pcmd, pins) \in proposals : pins = maxIns + 1\}$
59:             $decision \leftarrow deterministicFunction(commands)$
60:             $cmdsDecided[maxIns + 1] \leftarrow decision$
61:             $maxIns \leftarrow maxIns + 1$
62:         for $\forall(dmcd, nextMissing) \in decisions : cmdsDecided[nextMissing] = \perp$ do
63:             $cmdsDecided[nextMissing] \leftarrow dmcd$
```

of instance numbers $(ins, maxIns)$, a propose, and a decided message. The careful reader might notice that consensus instance numbers can grow infinitely large. Hence, messages can potentially have unbounded size. One option to avoid this, is to split a large message into multiple smaller ones and keep the exact same algorithm, with the slight change that it only considers a message when it has accepted all of its smaller messages. Another option is to explicitly state that messages consist of two parts, a header part that contains information such as instance numbers, signatures, etc., and an application part. Then, we can just ask to bound the application part of the messages, but not the header part [**?**]. We are not concerned with the header, which can grow so as to permit increasingly larger consensus instance numbers. On the other hand, this bound on the application part of a message is important to prevent "cheating" in the sense of batching all the commands from multiple consensus instances in a single message.

onReceive operates as follows. First, it checks whether the client sent a message (Line 41), and if so adds the sent command to *cmdsSet* (Line 42) and removes it from *responses* (Line 43). Then, the process checks if it is suspended in this round (Line 46), and if this is the case it does not perform any other operation. If the process is correct and $messageFor[client] \neq \perp$ (Line 47) it means that it successfully sent the previous response to the client, so it increases *lastResponse* (Line 48) and clears $messageFor[client]$ (Line 49). Then, it initializes *proposals* and *decisions* to be empty sets (Line 50). Then, it goes through the *responses* (Line 51) to update *maxIns* (Line 52), update *nextMissing* for each process (Line 53) and gather proposals and decisions from the responses (lines 54-55)[11]. Afterwards, the process adds its own proposal to the set of proposals (Line 56). Then, if there are proposals (Line 57) for instance number equal to $maxIns + 1$, all the commands are extracted from such proposals (Line 58) (note that we employ pattern matching by using the symbol _). The extracted commands are passed through some deterministic function (similar to Algorithm 1) and a value is decided upon (Line 60). Subsequently, *maxIns* is incremented appropriately (Line 61). At the end, the process goes through the decided messages (Line 62) and utilizes delivered commands that it needs.

Finally, note that the algorithm accepts optimizations (e.g., updating the *nextMissing* array after deciding on a command in Line 60), but we omitted them for clarity.

Proof. In what follows we prove that the algorithm satisfies safety (i.e., it is a valid SMR algorithm) and liveness (i.e., a clients eventually gets a response for any command it proposes) in theorems 8 and 12 respectively. We start by proving some useful lemmas. Note that in what follows we consider that the commands proposed by the client are always distinguishable. Furthermore, we denote with $variable_p[i]$ the value of $variable[i]$ of process p. Finally, when we state that a variable has a specific value in some round r, we refer to the beginning of round r (exactly before Line 14).

▶ **Lemma 4.** *For any $p \in \Pi$, $maxIns_p$ never decreases.*

Proof. Note that $maxIns_p$ is only modified in lines 52 and 61. In Line 52, $maxIns_p$ cannot decrease since it is updated to be the maximum of its own value and the *maxIns* received by some other process, so it will be greater or equal to what it was before. In Line 61, $maxIns_p$ is incremented by one, so again it does not decrease. ◀

▶ **Lemma 5.** *For any $p \in \Pi$ and $i \in \mathbb{N}^+$, if $i \geq maxIns_p + 1$, then $cmdsDecided_p[i] = \perp$.*

[11] Note that a response might not contain a proposal or a decision.

Proof. We use induction to prove this lemma. At the beginning of the first round, the property trivially holds. Assume it holds at the beginning of round r. We will show it holds at the beginning of round $r+1$. During the execution of round r there are two possibilities for $cmdsDecided_p$ to be modified so that the property will not hold. One is for $cmdsDecided_p$ to be written in Line 60 and another to be written in Line 63. If a write occurs in Line 60 the property does not hold but $maxIns_p$ is incremented immediately afterwards in Line 61, so the property holds back up again. The other case is that $cmdsDecided_p$ is written in Line 63. If $nextMissing = maxIns_p + 1$, then the property will not hold in round $r+1$. However, this would imply that some other replica sent a decided message with $maxIns_p + 1$ in the previous round. But then $maxIns_p$ in Line 52 would have been updated to correspond to this fact, a contradiction. Finally, note that due to Lemma 4 $maxIns_p$ never decreases, so the property cannot be circumvented by a reduction in $maxIns_p$. Therefore, the property holds at the beginning round $r+1$ and hence of every round. ◀

▶ **Lemma 6.** *For any $p \in \Pi$ and $i \in \mathbb{N}^+$, $cmdsDecided_p[i]$ is written at most once.*

Proof. We note that $cmdsDecided_p[i]$ for a specific $i \in \mathbb{N}^+$ is only updated in Algorithm 3 and in two places: Line 60 and Line 63. Furthermore, updates to $cmdsDecided_p$ take only place by processes that were not suspended in this round (Line 46). An update of $cmdsDecided_p$ in Line 60 occurs for the index $maxIns_p + 1$. Note that $maxIns_p$ since the start of the round might have only increased in Line 52, so the result of Lemma 5 is still satisfied. Due to Lemma 5, $cmdsDecided_p[maxIns_p+1]$ will be $\perp$. Hence updates in Line 60 can only occur in slots of $cmdsDecided_p$ that do not contain a command (i.e., a slot i such that $cmdsDecided_p[i] =\perp$). Therefore, slots that contain commands will not get overwritten in Line 60. Similarly, the update in Line 63 only occurs if $cmdsDecided_p[i] =\perp$ (where $i = nextMissing$) and not otherwise, so a $cmdsDecided_p[i]$ that already contains a command ($\neq\perp$) will not get updated in Line 63. Therefore, a specific position in $cmdsDecided_p$ gets updated at most once. ◀

▶ **Lemma 7.** *For any two processes $p, p' \in \Pi$ that are correct in a round r, then $maxIns_p = maxIns_{p'}$ immediately after Line 52 in round r.*

Proof. For this lemma, we use a similar argument to the one used to prove that the consensus Algorithm 1 satisfies agreement. All the correct processes would be able to deliver their local $maxIns$. Then each correct process will apply get the maximum for all the $maxIns$ values it delivered (Line 52), as well as its own. Hence, they will have the exact same value. ◀

▶ **Theorem 8.** *Algorithm 3 is a valid SMR algorithm.*

Proof. To show that the algorithm is valid, we have to show that the logs ($cmdsDecided$) are always consistent with each other.

We prove by induction that Algorithm 3 has the following property: for any two processes $p, p' \in \Pi$, if $cmdsDecided_p[i] \neq cmdsDecides_{p'}[i]$ for some $i \in \mathbb{N}^+$, then $cmdsDecided_p[i] =\perp$ or $cmdsDecided_{p'}[i] =\perp$. In the first round, the property trivially holds since $cmdsDecided_p[i] =\perp$ $\forall i \in \mathbb{N}^+$ and $\forall p \in \Pi$. Assume the property holds for all the rounds up to the r-th one. We will prove that it holds for round $r+1$. For this, we note that $cmdsDecided$ is updated in two different places, so we consider two cases:

- If a proposal takes place in Line 60. Due to Lemma 7, all correct processes will have the exact same $maxIns$ values, so they would all consider the exact same set of proposals (Line 58). Similar to the consensus Algorithm 1, correct processes will choose the exact same set of commands as well and pass them through the *deterministicFunction* to make a decision. Therefore, all replicas will store the exact same *decision* in their *cmdsDecided* log and hence the property still holds.

- If *cmdsDecided* is updated in Line 63, this means that the process received a decided message. This decided message was computed in a previous round (Line 37), but in the previous round the property holds (by induction). Therefore, if more than one process updates the *nextMissing* array, they will update it with the exact same command.

Finally, since the same log position is never updated more than once due to Lemma 6, the property will always be satisfied. Therefore, based on the definition of a valid SMR Algorithm (Section 2), Algorithm 3 is valid. ◀

▶ **Lemma 9.** *If $cmdsDecided_p[i] = cmd \neq \perp$ for some process $p \in \Pi$, then there are at least $n - 1$ processes with $cmdsDecided[i] = cmd$.*

Proof. In other words, this lemma states that each decided command exists in the log of $n - 1$ replicas. When a command is first decided upon in Line 60, every correct process at that round (and there are at least $n - 1$ correct processes in each round) decides on the exactly same command, since all correct processes see the exact same *maxIns* value (Lemma 7). Hence, at least $n - 1$ processes will store this command in *cmdsDecided*. ◀

▶ **Lemma 10.** *If $cmdsDecided_p[i] \neq \perp$, then eventually at least $n - 1$ processes, for all $j : 1 \leq j \leq i$ will have $cmdsDecided[j] \neq \perp$.*

Proof. Recall, that in every round, every process sends *ins* (Line 26) to the other processes informing them on the instance it is currently trying to get a command for. If the process is correct in the round, at least $n - 1$ of the other correct processes in this round will deliver the message and update their local *nextMissing* array (Line 53). From Lemma 9 we know that each decided command exists in the log of at least $n - 1$ processes. Since $n \geq 3$ and at most one process can be suspended in a round, we know that there will always be one correct process that contains a command that we are missing and that can help (Line 37). Therefore, in the next round if this process is correct it will receive a decision. Hence, eventually at least $n - 1$ processes will eventually fill up their log up to position i. ◀

▶ **Lemma 11.** *If a command c is decided ($\exists p \in \Pi$ and $\exists j \in \mathbb{N}^+$ such that $cmdsDecided_p[j] = c$), then the client eventually gets a response for command c.*

Proof. Due to Lemma 10, we know that if a command c appears in the log of some process, eventually c will appear in the logs of at least $n - 1$ processes, as well as all the previous commands in the log. Each process would have applied the command to the local state machine and stored the response in *clientResponses* (Line 23). Note however, that a process sends a response of a command to the client only if it is certain that the response to the previous command has been successfully delivered by the client. So only if the process is correct in a round and it can be assured that the previous response message was sent to the client (see lines 48 and 49), only then, the process sends the response to the next command to the client. Therefore, the client will eventually get a response for command c. ◀

▶ **Theorem 12.** *If a client suggests a command, then the client eventually gets a response.*

Proof. Assume by contradiction that the k-th command c_k is the first command that is suggested by the client in which the client never gets a response for. We will prove that the client will eventually get a response for c_k and hence a contradiction. First, note that since at least $n - 1$ processes are correct in each round, at least $n - 1$ processes will add

c_k to their log (Line 42). The client only suggests a command if it received a response for the previous command (see Section 2). Hence, processes have already decided on the c_{k-1} command. From Lemma 9, we know that c_{k-1} exists in the log of at least $n-1$ processes. Furthermore, due to Lemma 10, eventually all processes will fill their logs up to command c_{k-1}. This means that eventually *cmdsSet* will contain c_k as a the first command for at least $n-1$ processes, since all the other commands will be decided and removed from the set (Line 21 and 34). Therefore, c_k will be eventually proposed with instance number k, where $k = maxIns_p + 1$ for some process p, and hence it will be decided. Finally, from Lemma 11, since command c_k is decided, the client will eventually get a response for this command. ◀