Making Fast Consensus Generally Faster

[Technical Report]

Sebastiano Peluso
Virginia Tech
peluso@vt.edu

Alexandru Turcu
Virginia Tech
talex@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Giuliano Losa
Virginia Tech
giuliano.losa@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

***Abstract*—New multi-leader consensus protocols leverage the Generalized Consensus specification to enable low latency, even load balancing, and high parallelism. However these protocols introduce inherent costs with significant performance impact: they need quorums bigger than the minimum required to solve consensus, and need to track dependency relations among proposals. In this paper we present M^2PAXOS, an implementation of Generalized Consensus that provides fast decisions (i.e., delivery of a command in two communication delays) by leveraging quorums composed of a majority of nodes, and by exploiting workload locality. M^2PAXOS does not establish command dependencies based on conflicts, but it maps nodes to accessed objects, enforcing that commands accessing same objects are ordered by the same node. Our experimental evaluation confirms the effectiveness of M^2PAXOS, gaining up to 7× over state-of-the-art Consensus and Generalized Consensus algorithms under partitioned data accesses, and up to 5.5× using the TPC-C workload.**

I. Introduction

Paxos [1] is an algorithm for solving the consensus problem [2] in an asynchronous network, even in the presence of crashes, and is often used to build strongly consistent and fault-tolerance distributed services. In particular, Paxos can be leveraged for implementing practical strongly consistent and fault-tolerant transactional systems ([3], [4], [5], [6]) such as Google's Spanner [3]. Despite its widespread use, Paxos suffers from performance bottlenecks when deployed on networks with large amounts of nodes. For example, in its widely adopted and more practical deployment, i.e., Multi-Paxos [7], there is a designated leader which is responsible for ordering received commands, and allows the implementation of consensus in as few as three communication delays in crash-free executions. However, in practice, that leader constitutes a bottleneck that limits the performance of the whole system.

Several recent algorithms ([8], [9], [10]) eliminate the bottleneck constituted by the unique leader by allowing multiple nodes to operate as leaders at the same time. If this, on the one hand, gives the opportunity to balance the load and avoids a single point of decision (i.e., a designated leader), on the other hand it introduces the potentially high cost of handling contention among the various leaders issuing proposals concurrently. For example, EPaxos [8], a recent multi-leader consensus algorithm, employs four communication delays in order to safely decide a proposed command in case of contention.

To reduce the chances of contention among leaders, a common approach adopted by multi-leader algorithms is to relax the consistency requirement of Consensus, which demands that at most one proposal can be decided in a Consensus instantiation, by allowing that multiple proposals can be decided at the same time as long as their contents are not conflicting, i.e., they are commands whose executions commute according to the application semantics. This approach implements a more general variant of Consensus, called Generalized Consensus [11], [12], which has been proved to be sufficient for providing strong consistency in replicated services, since the outcome of the execution of a sequence of commutable commands on different nodes is independent of the order they are executed [11].

However, the advantages of Generalized Consensus implementations come at the cost of requiring synchronous communication with a larger set of nodes, additional computation for discriminating whether proposals are dependent, i.e., conflicting, or not, and bigger messages in order to include information about dependencies among proposals. Indeed, these algorithms reduce the minimum number of communication delays required to take a decision for one command from three to two in the absence of conflicts, but, to safely decide in two communication delays, a leader must communicate with a *fast quorum* of nodes [13], whereas Multi-Paxos needs only to communicate with a smaller *classic quorum* of nodes. Classic quorums are often composed of $\left\lfloor \frac{N}{2} \right\rfloor + 1$ nodes, where N is the total number of nodes, whereas fast quorums must be composed of at least $\left\lfloor \frac{2 \cdot N}{3} \right\rfloor + 1$ nodes. Thus, Generalized Consensus algorithms and (Multi-)Paxos choose different tradeoffs between size of quorums and communication delays. Moreover, current Generalized Consensus algorithms must compute dependency relations among commands, a potentially costly operation, and must exchange them among nodes, generating a higher bandwidth usage. In contrast, (Multi-)Paxos does not use dependency relations at all.

Existing theoretical results on the cost of implementing consensus [14] prove that one cannot achieve an optimal tradeoff which combines both the adoption of classical quorums and decisions in two communication delays in all the possible executions; also it is not known whether the costs associated with having dependency relations can be avoided. In this paper we circumvent this restrictions by investigating the feasibility of having minimal size of quorums, low delay, and no dependency relations under common application workloads. In other words, we aim at answering the following question: can we guarantee a *generally faster* performance at the cost of having a slightly more expensive decision process only in case the application exhibits unfavorable access patterns? Our contribution proves that under a workload in which two different nodes do not often propose conflicting commands, which is common in scalable transactional systems [15], [16], [17], one can combine the advantages of multi-leader Generalized Consensus algorithms and Multi-Paxos, i.e., obtaining load balancing among nodes, a high proportion of decisions in two communication delays, the adoption of classic quorums, and no dependency relations to compute or exchange.

We present M^2PAXOS[1], an implementation of Generalized Consensus that *generally*, which in this paper means under favorable conditions of low inter-node contention and temporal locality, where a node likely issues commands on objects already accessed in the past, provides the following optimal features: M^2PAXOS decides commands in only two communication delays; it does not compute dependencies on commands, and hence it does not exchange dependencies among nodes; it relies on classic quorums of size equal to $\lfloor \frac{N}{2} \rfloor + 1$, like Multi-Paxos. We name the aforementioned workload as *partitionable*.

Underlying M^2PAXOS lies the following observation: Generalized Consensus algorithms conservatively use fast quorums and dependency relations because they must recover when interfering commands are ordered differently by some nodes after an attempted fast decision. However, if we can prevent different nodes from issuing conflicting commands, then we can reduce the inherent costs of those algorithms.

M^2PAXOS is designed exploiting the above intuition: it ensures that conflicting commands are ordered in the same way in all nodes by requiring that, on the proposal of a command c, a leader first acquires the *exclusive ownership* of all the commands interfering with c, called the interference set of c, before trying to decide c. Acquiring exclusive ownership of interference sets prevents contention among different leaders: any two conflicting commands will be either ordered by a unique leader, namely their owner, or they will be separated by a change of ownership. Once the ownership of interference sets is stable in the system, commands can be ordered in two message delays in parallel in case they are assigned to different leaders. To simplify the ownership acquisition, we assume that the semantics of the commands is given in terms of the set of objects that they access. Therefore, we can over-approximate the interference set of a command c by the set of all commands which access at least one object accessed by c as well.

To implement exclusive object ownership, M^2PAXOS adapts the mechanism used by Multi-Paxos to ensure that there is only one leader at a time: M^2PAXOS can be seen as running one incarnation of Multi-Paxos per object, under the restriction that a node only accepts a command c if it can do so in all incarnations of Multi-Paxos corresponding to c's accessed objects. Thus, a node successfully orders a command c only if it is the leader, in the sense of Multi-Paxos, of all the incarnations corresponding to the objects accessed by c.

M^2PAXOS manages object ownership to order a command c as follows: *i)* if the proposer of c has the ownership of all the objects accessed by c, it orders c as the next command to execute on those objects, in two communication delays; *ii)* if the proposer of c does not have the necessary ownerships for c, but there is another node that has them, then M^2PAXOS forwards c to that node, thus adding one communication delay to the previous case; and *iii)* in all the other cases, the node proposing c first acquires the ownership of c's objects in all the incarnations of Multi-Paxos corresponding to the objects accessed by c, using the same mechanism as Multi-Paxos, and then performs step *i)*.

[1]A poster version of this paper recently appeared. Its venue has been hidden to respect the double blind rules, as indicated by the DSN Program Chairs.

M^2PAXOS is particularly effective in deployments where the set of accessed objects is well defined once a "home" object is accessed, e.g., the access pattern of the well known TPC-C benchmark [18] involves first an access to a `warehouse` (i.e., the home object) and then the subsequent accessed objects will be very likely related to that `warehouse`.

We implemented M^2PAXOS in the Go programming language and compared against Generalized Paxos [11], Multi-Paxos [7], and EPaxos [8], a recent high performance implementation of generalized consensus. M^2PAXOS is simple: there is no time consuming operation performed on its critical path and it scales well in partitioned workloads. Once the ownership is defined and is stable, M^2PAXOS substantially outperforms all competitors. The maximum speed-up observed against EPaxos, which is the best competitor, is $7\times$ when 49 nodes are deployed and objects are partitioned across them. We evaluated M^2PAXOS also by implementing a benchmark producing the TPC-C workload. In this deployment, M^2PAXOS outperforms EPaxos by as much as $5.5\times$ and Multi-Paxos by as much as $2.5\times$.

The implementation of M^2PAXOS is publicly available and a link to the sources will be disclosed once the anonymous review period is over. Moreover, we have formalized a high level description of M^2PAXOS in TLA+ [19] and have model-checked it with the TLC model-checker, obtaining high confidence that our protocol is correct. The TLA+ formalization can be found in the appendix.

II. Related Work

In the classic Paxos algorithm, a value is learned after a minimum of four communication delays. Progress guarantees are provided as long as there are no two nodes trying to become leaders concurrently (this step is called Prepare phase). Multi-Paxos alleviates this problem by letting a Prepare phase cover an entire sequence of values. This effectively establishes a proposer that acts as a designated leader. Once the leader is elected, new values can be learned in only three communication delays, and progress can be guaranteed in periods of synchrony. Fast Paxos [13] can eliminate one communication delay by having proposers bypass the leader and broadcast their requests directly to nodes, which is called a fast path. If a fast path fails due to concurrent proposals (called a collision), the designated leader needs to take over the decision by adding two additional communication delays. Moreover, acceptors in Fast Paxos have to wait for a number of replies that is greater than a majority of nodes in the fast rounds (a minimum of $\lfloor \frac{2 \cdot N}{3} \rfloor + 1$, a fast quorum).

Generalized Paxos [11] solves Generalized Consensus and, as Fast Paxos, it can decide commands in two communication delays. Unlike Fast Paxos, it can do that also in the case of concurrent proposals as long as commands are commutative. If not, a recovery from a collision scenario requires the same costs paid by Fast Paxos. This overhead is avoided by the Fast Genuine Generalized Consensus (FGGC) algorithm [20], which is able to reduce the extra communication delays for the recovery from four to one by leveraging the following assumption: every fast quorum in a round has to include the leader of that round. FGGC is optimized to provide reasonable performance also in case of high and non-well-partitioned contention scenarios (unlike M^2PAXOS), but it may suffer from higher latency because nodes have to wait for the leader in all rounds.

EPaxos [8] is a multi-leader solution to the generalized consensus problem. EPaxos employs dependency tracking and fast quorums to deliver non-conflicting commands using a fast path of two communication delays. In the presence of conflicts however, the protocol takes a slow path of four communication delays before delivering.

The advantages of M^2PAXOS over the previous Paxos-based algorithms are clear: M^2PAXOS is able to decide commands in two communication delays, as they do, but without relying on either fast quorums, a designated leader, or exchanging and processing dependencies among commands.

M^2PAXOS is also related to the Asynchronous Lease-based Certification protocol (ALC) [21] and LILAC-TM protocol [22] because they share the basic idea of exploiting ownership of objects to save communication steps during a distributed coordination. As we will show in Section IV-C, ALC and Lilac-TM address orthogonal problems whose solutions can be integrated in M^2PAXOS to boost its performance.

III. SYSTEM MODEL AND CONSENSUS

We assume a set of nodes $\Pi = \{p_1, p_2, \ldots, p_N\}$ communicating through message passing where messages may experience an arbitrarily long, although finite, delays and they do not have access to either a shared memory or a global clock. Nodes may fail by crashing, but do not behave maliciously. A node that does not crash is called correct; otherwise it is faulty. Because of the well-known FLP result [23], we assume that the system can be enhanced with the weakest type of unreliable failure detector [24] that is necessary to implement a leader election service [25]. The leader election (and thus the failure detector) is needed by M^2PAXOS to accomplish a successful change of object ownership if no conflicting commands are proposed in parallel. In addition, due to the result in [2], we assume that at least a strict majority of nodes, i.e., $\left\lfloor \frac{N}{2} \right\rfloor + 1$, is correct and thus at most $f = \left\lfloor \frac{N}{2} \right\rfloor$ nodes can be faulty at any time (as in Paxos).

We follow the definition of Generalized Consensus as in [11], where each node can propose commands for a set Cmd via C-PROPOSE($Cmd\ c$) interface, and nodes decide command structures $C\text{-}structs$ via C-DECIDE($C\text{-}struct\ cs$) interface. The specification is such that: commands that are included in the decided $C\text{-}structs$ must have been proposed (*Non-triviality*); if a node decided a $C\text{-}struct\ v$ at any time, then at all later times it can only decide $v \bullet \sigma$, where σ is a sequence of commands (*Stability*); and two $C\text{-}structs$ decided by two different nodes are prefixes of the same $C\text{-}struct$ (*Consistency*). Since the *Liveness* property of M^2PAXOS depends on the success of the object ownership acquisition, we adopt the following definition: if a command c has been proposed by a correct node and there is no other concurrent and conflicting command with c in the system, c will be eventually decided in some $C\text{-}struct$.

Finally we assume that commands are defined for accessing a set of objects whose identifiers are in the set LS. Therefore a command c is associated with a set of identifiers $c.LS \subseteq LS$.

IV. BUILDING THE PROTOCOL

Before going into the details of the protocol, in this section we give an overview of all its core parts, providing an intuition on how they work together and what role they play in the process of reaching consensus. We first describe how M^2PAXOS is able to provide the fastest delivery (Section IV-A), given the best partitionable workload, by reaching consensus in two communication delays and by relying on the existence of classic quorums of minimal size equal to $\left\lfloor \frac{N}{2} \right\rfloor + 1$, where at most $\left\lfloor \frac{N}{2} \right\rfloor$ nodes can be faulty. That is achievable in the optimal conditions for M^2PAXOS, namely when there are no conflicts among commands that are proposed by different nodes (as in other implementations of Generalized Consensus [11], [8]), and the application layer using the consensus service exhibits locality.

On the contrary, if different nodes submit conflicting commands concurrently, M^2PAXOS switches to a slower path of execution, whose length still depends on the characteristic of the conflicts. In particular, if a command submitted by a node only exhibits conflicts with commands submitted by at most one other node, M^2PAXOS guarantees to reach a consensus in three communication delays in a fault-free case (Section IV-B). That is a very appealing result because of the following twofold reason: on the one hand, we are able to meet the lower bound defined for the problem of consensus in an asynchronous system and in the presence of conflicts; on the other hand, unlike Fast/Generalized Paxos, we do not pay any additional overhead by switching from a fast path to a slower path, a feature that is important for the effectiveness of relying on fast decisions, as pointed out by the work in [26].

Finally, if the workload exhibits generic conflict patterns (Section IV-C), i.e., a command submitted by a node can conflict with commands submitted by multiple nodes, M^2PAXOS reaches consensus by paying a cost that can vary from the best case of four communication delays to the worst case of an unbounded number of communication delays. Even though the worst case does not seem acceptable, it does not result in a real limitation for M^2PAXOS because that basically happens in scenarios where it is not worth having a protocol optimized for low inter-node conflict rates (e.g., EPaxos [8], Fast/Generalized Paxos [11], M^2PAXOS), hence where adopting a classical Paxos implementation is more effective [27]. However as we will detail in Section IV-C, in this adverse scenario, M^2PAXOS can be integrated with other consensus implementations that give higher liveness guarantees, such as Paxos itself, by relying on techniques to switch among different protocols as in [28].

In the description of the protocol we do not explicitly refer to a phase that recovers from a crash in order to finalize the decision of commands that are proposed by the crashed nodes. Indeed we show that this recovery is embedded into the process of changing the ownership on an object l, because that change has to first take into account any pending command already accepted and not yet decided for l.

A. The Fastest Delivery

In M^2PAXOS we consider the problem of solving consensus using a different approach from the one considered so far by other scalable implementations of Generalized Consensus, e.g., EPaxos and Alvin. In those existing solutions a command is associated with a node (i.e., the command's leader), which is in charge of coordinating with the other nodes to define the command's position in the final sequence delivered by the consensus. On the contrary, in M^2PAXOS we map accessed objects to nodes. More formally, for the purpose of the presentation, we define a boolean function, $IsOwner : T \times \Pi \times LS \rightarrow bool$, where $IsOwner(t, p_i, l)$ returns $true$ if node p_i is the owner of object l at time t; otherwise $false$. This function is such that if $IsOwner(t, p_i, l) = true$, then $\forall p_j \neq p_i, IsOwner(t, p_j, l) = false$. For simplicity, hereafter we use the notation $IsOwner(p_i, l)$ to indicate $IsOwner(t, p_i, l)$ with t equal to the invocation time of $IsOwner$.

We also define a relation $Decided = LS \times IN$ that associates *objects* with *delivery positions*. In particular $\langle l, in \rangle \in Decided$ means that a command accessing object l has been decided after all the commands such that $\langle l, in' \rangle \in Decided \wedge in' < in$ and before all the commands such that $\langle l, in'' \rangle \in Decided \wedge in'' > in$.

If p_i is the proposer of a command c and $\forall l \in c.LS$ $IsOwner(p_i, l) = true$ (i.e., p_i is the owner of the objects accessed by c), then M^2PAXOS can solve consensus in two communication delays (*fast decision*) by relying on quorums of size equal to $\left\lfloor \frac{N}{2} \right\rfloor + 1$. In fact, informally, no other node can decide at the same time the order of some command accessing some (or all) objects requested by c given that p_i is the exclusive owner of those objects; and in order to guarantee recoverability a majority of nodes have to receive c.

This fast decision is simple and it proceeds as follows: triggering a C-PROPOSE(c) on p_i for a command c entails: *i)* broadcasting an ACCEPT with a pair $\langle l, in \rangle$ for each $l \in c.LS$, such that in is the minimum not yet decided position for l, i.e., $\langle l, in \rangle \notin Decided$; and then *ii)* waiting for a quorum of $\left\lfloor \frac{N}{2} \right\rfloor + 1$ ACK messages. The wait condition is necessary for the recoverability of the decision in case of faults. In fact, if p_i crashes after having taken the decision for c we are sure that there is at least one correct node in the system that observed that decision. We say that p_i is the leader of the *consensus instance* in of l.

As an example, let us consider two commands $c1$ and $c2$ proposed to the consensus layer and accessing the pairs of objects $\{A, B\}$ and $\{B, C\}$ respectively. Further, let us suppose that $c1$ was decided in position 1 for both objects A and B, and $c2$ was decided in position 2 for the object B and in position 1 for the object C. Therefore the sequence delivered by the consensus so far is $c1 \bullet c2$. At this point, let us consider $IsOwner(p_i, A) = true$ and $IsOwner(p_i, B) = true$, and p_i proposes command $c3$ such that $c3.LS = \{A, B\}$, i.e., $c3$ will access both A and B. Then p_i can simply broadcast an ACCEPT message for $c3$ with the set $ins = \{\langle A, 2 \rangle, \langle B, 3 \rangle\}$, meaning that it requests the other nodes to accept $c3$ as the command that follows both $c1$ and $c2$ in the final sequence.

When a node receives an ACCEPT message, it can broadcast an *ACK* in order to indicate that it is accepting to deliver c in the consensus instances specified by the received ACCEPT message. Afterwards, when a node receives a quorum of *ACK* messages for c, it can consider c as ready to be delivered in those consensus instances. In this scenario, which is clearly optimal for M^2PAXOS, we are able to decide a command in two communication delays by using a classic quorum size. As it will be clear later, that is not always the case because a node could also reply with a *NACK* message on the reception of an ACCEPT message. This can happen when a different node p_j wants to concurrently propose a command accessing part of (or all) the objects accessed by c. As a result, if the workload is partitionable, then a node will generally issue *ACK* rather than *NACK* as reply to ACCEPT messages, thus generally enabling fast deliveries.

B. Time to Forward

In addition to the previous case, a node p_i proposing a command c could not have the ownership on all the objects in $c.LS$. However, there could exist a different node p_j such that $\forall l \in c.LS$ $IsOwner(p_j, l) = true$. In this case, we can opt to forward command c to node p_j and rely on p_j for a fast delivery of c. For instance, this is what a propose phase does in Multi-Paxos, which relies on a designated leader to define the command to be decided in the next consensus instance.

Upon a C-PROPOSE(c) event on node p_i, p_i forwards c to p_j. This forwarding step triggers a new C-PROPOSE(c) event on node p_j, which can execute the steps of a fast decision for c as described in Section IV-A. We have to note that in this case, even though the command c cannot be decided in two communication delays, it is still decided in only three communication delays, which is the minimum cost due to solve consensus in an asynchronous system in case of concurrent conflicting proposals [14].

C. Reshuffling the Ownership

Finally, the application workload may generate a scenario where neither a proposer of a command c nor any other node in the system has the ownership of all the objects in $c.LS$. Therefore, in this case, M^2PAXOS needs to reshuffle the object ownerships such that one of the scenarios presented in Sections IV-A and IV-B is recreated. Here we opt to assign ownerships in a way such that the proposer of c will become the owner of all objects accessed by c. In fact, in that way we will have the chance of deciding c and subsequent commands that access objects in $c.LS$ in two communication delays.

Therefore, the proposer of c, say p_i, attempts to become the new owner for the next available consensus instances on all the objects in $c.LS$. We can choose to do this in different ways, according to the degree of conflicts. We propose a simple and generally effective way to reshuffle object ownerships, but that does not provide any guarantee on the maximum number of communication delays to be paid. Then we give some hints on how to solve this problem in a bounded number of communication delays. In this paper we do not focus on defining optimized policies that regulate when an object ownership is better to change because we believe it is an orthogonal problem and there are other more complex and effective solutions available (e.g., [22]). In our implementation we use a simple on-demand policy that attempts to change the ownership when a request is issued by the application.

The simple solution. When a node p_i has to propose a command c and there is no unique owner p_j (possibly equal to p_i) of all objects in $c.LS$, then p_i executes a Paxos prepare phase [1] in order to start a new epoch for the next available instances of all the objects in $c.LS$. The idea is the same adopted by Multi-Paxos when it elects a new leader in a new epoch e that will be responsible for solving consensus for all the proposals in e.

Let us say p_i wants to reach consensus for a command c in the next positions available for the objects in $c.LS$. Then it broadcasts a PREPARE message containing tuples $\langle l, in, e \rangle$, for each $l \in c.LS$, and such that in is the smallest instance associated with l where $\langle l, in \rangle \notin Decided$, and e is the successor of the current epoch number associated with l.

Afterwards, p_i waits for a quorum of replies and, if the quorum does not contain any *NACK* message, p_i has been acknowledged to be the current leader for all the objects in $c.LS$. At that point, p_i can just request the acceptance of c for all the positions in defined above, as explained in Section IV-A. That happens only if the received acknowledgements do not suggest the acceptance of any other command different from c. In fact, as it will be clearer in the next section, there could already be another command c' accepted in some of the positions selected by p_i but whose decision was not finalized yet. Such a scenario occurs if another node lost the ownership on some objects of c' after having sent an accept message for c'. We will address this case in Section V. On the other hand, if p_i receives at least one *NACK* in this phase, it is forced to retry the ownership acquisition with greater epoch numbers.

Bounding the Communication Delays. Negative acknowledgements received during the ownership acquisition could generate an unbounded sequence of restarts of the acquisition itself. This is not an optimal scenario for M^2PAXOS because this happens in case multiple nodes try to concurrently acquire the ownership on common objects. Typically, if the frequency of such attempts is high, it means that the workload using M^2PAXOS is not partitionable. However, even though we did not design the protocol to provide high performance in this type of workload, the correctness of M^2PAXOS is still preserved regardless of the cost of multiple attempts.

In case we would like to establish a bound on the number of communication delays paid in this phase, we can either totally order ownership acquisition requests, by relying on another separate consensus instance, or designate one single leader to be responsible for solving conflicts on ownership acquisitions. Also, to keep the performance consistent across varying workloads, we could use the approach described in [28] to combine M^2PAXOS with algorithms that perform well on workloads not favorable to M^2PAXOS. For example, we could obtain an algorithm that dynamically switches between M^2PAXOS and MultiPaxos according to the workload characteristics.

V. M^2PAXOS: PROTOCOL DETAILS

Since M^2PAXOS implements the Generalized Consensus specification, it exposes the interface C-PROPOSE($Cmd\ c$) used by any node to propose a command c, and the interface C-DECIDE($C\text{-}structs\ cs$) to deliver a $C\text{-}structs\ cs$ to any node. Before describing the details of the protocol, we introduce all the data structures associated with a node in the system. Then we will present the complete protocol, also covering all the aspects of which we only provided an intuition in Section IV.

A. Data Structures

Each node p_i maintains the following data structures:

- $Decided$ and $LastDecided$. The former is a multidimensional array that maps a pair of ⟨object,consensus instance⟩ to a command. $Decided[l][in] = c$ if c has been decided in the consensus instance in (i.e., in position in) of the object l. The latter is a unidimensional array that maps an object to consensus instance, and such that $LastDecided[l] = in$ if in is the most recent instance for which p_i has observed a decision for object l. The initial values are $NULL$ in $Decided$, and they are 0 in $LastDecided$.
- $Epoch$. It is an array that maps an object to an epoch number (i.e., a non-negative integer). $Epoch[l] = e$ means that e is the current epoch number that has been observed by p_i for the the object l. The initial values are 0.
- $Owners$. It is an array that maps an object to a node. $Owners[l] = p_j$ means that p_j is the current owner of the object l. The initial values are $NULL$.
- Rnd, $Rdec$ and $Vdec$. They are three multidimensional arrays. The first two map a pair of ⟨object,consensus instance⟩ to an epoch number; the third one maps a pair of the form ⟨object,consensus instance⟩ to a command. In particular, $Rnd[l][in] = e$ if e is the highest epoch number in which p_i has participated in the consensus instance in of object l; $Rdec[l][in] = e$ if e is the highest epoch number in which p_i has accepted a command for the consensus instance in of object l; and $Vdec[l][in] = c$ if c is the command accepted by p_i in the epoch $Rdec[l][in]$ for the consensus instance in of object l. The initial values in Rnd and $Rdec$ are 0, while the ones in $Vdec$ are $NULL$.
- $Acks$. It is a multidimensional array used to collect the ACKACCEPT messages sent as a reply to an ACCEPT message. The pair $\langle c, j \rangle$ is in the set $Acks[l][in][e]$ iff p_i has received a ACKACCEPT with command c for the consensus instance in of the object l in the epoch e.
- $Cstructs$. It is the most recent value of the *command structures* delivered by p_i. Its initial value is $\perp$.

B. The Protocol

A command submitted to M^2PAXOS via the C-PROPOSE($Cmd\ c$) goes throughout 4 phases: *i)* the *Coordination phase*, whose pseudocode is presented in Algorithm 1, which establishes whether the command can be decided in two, three or more communication delays; *ii)* the *Accept phase*, whose pseudocode is presented in Algorithm 2, which requests the acceptance of the command in a certain position with respect to the other submitted commands; *iii)* the *Decision phase*, whose pseudocode is presented in Algorithm 3, which decides the command's final position, appends the command to the next $Cstructs$ to be delivered, and executes the delivery of the $Cstructs$; and *iv)* the *Acquisition phase*, whose pseudocode is presented in Algorithm 4, which executes a reconfiguration of the ownership, if needed, in order to elect the node in charge of requesting the acceptance of the command.

1) Coordination phase (Algorithm 1): When a command c is proposed by node p_i via C-PROPOSE($Cmd\ c$), M^2PAXOS coordinates the decision for c. For each object l in $c.LS$ such that there is no position in that is associated with l and was decided for c, it adds the next available position for l, i.e.,

$LastDecided[l]+1$, to the ins set (line 2). Therefore if ins contains the pair $\langle l, in\rangle$, we say that p_i wants to participate to decide c in the consensus instance in for l. In other words, M^2PAXOS tries to deliver command c after all the commands c' such that $\exists\langle l, in\rangle \in ins$ and $Decided[l][in'] = c'$ for some $in' < in$.

Clearly, if ins is an empty set, M^2PAXOS does not execute any further step, because it already found a delivery position for c. Otherwise it distinguishes three cases depending on whether c can be decided in two or three communication delays, or we need a reconfiguration of the ownership relation.

In the first case, if for each pair $\langle l, in\rangle$ in ins, p_i is the current owner of object l (lines 5 and 19–23), p_i can execute an *Accept phase* for command c in positions ins without changing the epochs for those positions (lines 6–8). If that phase succeeds then c will be eventually delivered by all correct nodes in two communication delays; otherwise p_i restarts the *Coordination phase* (lines 9–10). We notice that the value of the first input parameter of the *AcceptPhase* function is an empty array because in this case the node p_i does not request the acceptance of any other command different from c. As we will explain in detail in Section V-B2, there are scenarios in which p_i must prioritize the acceptance of commands different from c.

In the second case, if for each pair $\langle l, in\rangle$ in ins, p_k (where $k \neq i$) is the current owner of object l (lines 11 and 25–29), p_i can request the execution of the *Coordination phase* for c to p_k (lines 12–13). In the best case, p_k will execute lines 1–8 by reaching a decision in two communication delays for c, so by paying a total cost of three communication delays if we take into account the forward of c from p_i to p_k. However, to avoid blocking conditions (e.g., if p_k crashed, and p_i did not detect the crash), if p_i does not observe c as decided in at least one position in for each object l in $c.LS$ when a configurable timeout expires (line 13), p_i takes over and re-executes the *Coordination phase* (lines 14–15).

In the third case (lines 16–17), neither p_i nor any other node p_k other than p_i have the necessary ownership to execute the *Accept phase* for c. Therefore, p_i forces a reconfiguration of the ownership by entering the *Acquisition phase*. So p_i tries to acquire the ownership on $c.LS$ and, as we will explain in Section V-B4, it also executes the *Accept phase*.

2) Accept phase (Algorithm 2): In this phase p_i requests the acceptance of a command in all the positions listed in ins for the epochs in eps to a quorum of nodes (lines 8–9). In the case where this phase starts at line 8 of Algorithm 1, the command that is broadcast by p_i is c, namely the command that p_i is proposing (lines 5–7). Otherwise, this is an *Accept phase* called during an *Acquisition phase*, and p_i needs to take into account the outcome of the ownership reconfiguration, i.e., $toForce$, to decide the command to be accepted.

Even though this last case will be clearer when we will analyze the *Acquisition phase* in Section V-B4, we have to take into account that the current *Accept phase* executed by p_i after having acquired the necessary ownership for c, could follow a concurrent *Accept phase* executed by another node p_k for a command c' conflicting with c. In that case, if there is some node that already accepted c' for a certain pair $\langle l, in\rangle \in ins$,

Algorithm 1 M^2PAXOS: *Coordination phase* (node p_i).

```
1:  upon C-PROPOSE(Cmd c)
2:      Set ins ← {⟨l, LastDecided[l] + 1⟩ : l ∈ c.LS ∧ ∄in : Decided[l][in] = c}
3:      if ins = ∅ then
4:          return
5:      if ISOWNER(p_i, ins) = ⊤ ∧ ∄⟨l, in⟩ ∈ ins : Vdec[l][in] ≠ NULL then
6:          Array eps
7:          ∀⟨l, in⟩ ∈ ins, eps[l][in] ← Epoch[l]
8:          Bool acc ← AcceptPhase({}{}, c, ins, eps)
9:          if acc = ⊥ then
10:             trigger C-PROPOSE(c) to p_i
11:     else if |GETOWNERS(ins)| = 1 ∧ p_i ∉ GETOWNERS(ins) then
12:         send PROPOSE(c) to p_k ∈ GETOWNERS(ins)
13:         wait(timeout) until ∀l ∈ c.LS, ∃in : Decided[l][in] = c
14:         if ∃l ∈ c.LS, ∄in : Decided[l][in] = c then
15:             trigger C-PROPOSE(c) to p_i
16:     else
17:         ACQUISITIONPHASE(c)
18:
19: function Bool ISOWNER(Node p_i, Set ins)
20:     for all ⟨l, in⟩ ∈ ins do
21:         if Owners[l] ≠ p_i then
22:             return ⊥
23:     return ⊤
24:
25: function Set GETOWNERS(Set ins)
26:     Set res ← ∅
27:     for all ⟨l, in⟩ ∈ ins do
28:         res ← res ∪ {Owners[l]}
29:     return res
```

p_i cannot ignore that, and it has to collaborate for the decision of c' in position in (lines 3–4).

This phase can abort by returning $\perp$ after having broadcast the ACCEPT message if p_i receives at least one negative reply, i.e., an ACKACCEPT message marked as $NACK$ (lines 10–11). Indeed, when a node receives an ACCEPT message for a set of commands $toDecide$, a set ins of pairs $\langle l, in\rangle$, and an array epochs eps (line 16), it can reply with a $NACK$ if there exists at least an object l and a position in, such that $\langle l, in\rangle \in ins$, and the node already participated in the consensus instance in for l by using an epoch number greater than $eps[l][in]$ (lines 23–24). This can obviously happen when there is another node that is concurrently trying to propose another command in position in for l.

Otherwise, if that is not the case (line 17–22), the node can broadcast an ACKACCEPT message with ACK, and for each $\langle l, in\rangle \in ins$ it also changes the following information: the current owner of l is the sender of the ACCEPT (line 18); the last command accepted in $\langle l, in\rangle$ is $toDecide[l][in]$ (line 19); the greatest epoch in which the node has accepted a value in $\langle l, in\rangle$ is $eps[l][in]$ (line 20); and the greatest epoch in which the node has participated for the consensus instance $\langle l, in\rangle$ is $eps[l][in]$ (line 21).

Therefore if p_i receives at least a quorum of ACKACCEPT messages marked as ACK for the commands in $toDecide$, it can broadcast the final decision $toDecide$ via a DECIDE message (lines 12–14).

3) Decision phase (Algorithm 3): In this phase a node p_i can simply mark a command c as decided in position in for a certain object l accessed by c, by setting $Decided[l][in]$ to c (lines 4 and 10). This happens in the following two cases.

First, p_i received a DECIDE message for commands $toDecide$ and positions ins, such that $\langle l, in\rangle \in ins$ and

Algorithm 2 M^2PAXOS: *Accept phase* (node p_i).

```
1: function Bool ACCEPTPHASE(Array toForce, Cmd c, Set ins, Array eps)
2:     Array toDecide
3:     for all ⟨l, in⟩ ∈ ins : toForce[l][in] = ⟨c′, −⟩ : c′ ≠ NULL do
4:         toDecide[l][in] ← c′
5:     if ∀⟨l, in⟩ ∈ ins, toDecide[l][in] = NULL then
6:         for all ⟨l, in⟩ ∈ ins do
7:             toDecide[l][in] ← c
8:     send ACCEPT(⟨toDecide, ins, eps⟩) to all p_k ∈ Π
9:     Set replies ← receive ACKACCEPT(⟨ins, eps, toDecide, −⟩) from Quorum
10:    if ∃⟨ins, eps, toDecide, NACK⟩ ∈ replies then
11:        return ⊥
12:    else
13:        send DECIDE(⟨toDecide, ins, eps⟩) to all p_k ∈ Π
14:        return ⊤
15:
16: upon ACCEPT(⟨Array toDecide, int ins, Array eps⟩) from p_j
17:    if ∀⟨l, in⟩ ∈ ins, Rnd[l][in] ≤ eps[l][in] then
18:        ∀⟨l, in⟩ ∈ ins, Owners[l] ← p_j
19:        ∀⟨l, in⟩ ∈ ins, Vdec[l][in] ← toDecide[l][in]
20:        ∀⟨l, in⟩ ∈ ins, Rdec[l][in] ← eps[l][in]
21:        ∀⟨l, in⟩ ∈ ins, Rnd[l][in] ← eps[l][in]
22:        send ACKACCEPT(⟨ins, eps, toDecide, ACK⟩) to all p_k ∈ Π
23:    else
24:        send ACKACCEPT(⟨ins, eps, toDecide, NACK⟩) to p_j
```

$toDecide[l][in] = c$ (lines 1–4). Second, p_i received at least a quorum of ACKACCEPT messages marked as ACK for commands $toDecide$, positions ins and epochs eps, such that $\langle l, in\rangle \in ins$ and $toDecide[l][in] = c$ (lines 6–10).

Furthermore, as soon as there exists a command c such that c has been decided in some position in associated with an object l for all $l \in c.LS$, M^2PAXOS checks whether it can append c to the next $Cstructs$ to be delivered. That is true if c immediately follows the last appended message for each object l in $c.LS$ (line 12). Then, once a new command has been appended to the $Cstructs$ (line 13), p_i triggers the delivery of the updated $Cstructs$ (line 14), and it advances the pointers to the last appended messages (lines 15–16).

Algorithm 3 M^2PAXOS: *Decision phase* (node p_i).

```
1: upon DECIDE(⟨Set toDecide, Set ins, Array eps⟩) from p_j
2:     for all ⟨l, in⟩ ∈ ins do
3:         if Decided[l][in] = NULL then
4:             Decided[l][in] ← toDecide[l][in]
5:
6: upon ACKACCEPT(⟨Set ins, Array eps, Array toDecide, ACK⟩) from p_j
7:     for all ⟨l, in⟩ ∈ ins do
8:         Set Acks[l][in][eps[l][in]] ← Acks[l][in][eps[l][in]] ∪
                                           ∪ {⟨toDecide[l][in], j⟩}
9:         if |Acks[l][in][eps[l][in]]| ≥ sizeof(Quorum) ∧
                                       ∧ Decided[l][in] = NULL then
10:            Decided[l][in] ← c : ⟨c, −⟩ ∈ Acks[l][in][eps[l][in]]
11:
12: upon (∃c : ∀l ∈ c.LS, ∃in : Decided[l][in] = c ∧
                                       ∧ in = LastDecided[l] + 1)
13:    Cstructs ← Cstructs • c
14:    trigger C-DECIDE(Cstructs)
15:    for all l ∈ c.LS do
16:        p_i.lastDecided[l] + +
```

4) Acquisition phase (Algorithm 4): A node p_i tries to acquire the necessary ownership to decide command c (line 1). For each object l in $c.LS$ such that there is no position in that is associated with l and was decided for c, p_i adds the next available position for l, i.e., $LastDecided[l] + 1$, to the ins set (line 2). Further, for each pair $\langle l, in\rangle \in ins$, it increments the current epoch number for l (lines 3–4). Then it broadcasts a PREPARE message with ins and the new epochs eps, and waits for a quorum of ACKPREPARE replies (lines 5–6).

If at least one received ACKPREPARE is marked as $NACK$ (line 7) then the ownership acquisition did not succeed and p_i restarts a new *Coordination phase* for c by calling C-PROPOSE(c) (line 8). To guarantee that c is eventually decided also in scenarios of high conflict, p_i might also decide to trigger C-PROPOSE(c) on a designated leader by switching to a classic Paxos protocol as described in Section IV-C.

A node can refuse a received PREPARE message (line 15) by replying with an ACKPREPARE marked as $NACK$, in case there exists a position $\langle l, in\rangle$ in the received ins set such that the received $eps[l][in]$ does not move any epoch forward on that node, i.e., $eps[l][in] \le Rnd[l][in]$ (lines 20–21). Rather, if that is not the case, the node replies with an ACKPREPARE marked as ACK, by including the last epoch in which it accepted a command and the last command accepted for any position $\langle l, in\rangle$ in the received ins. It also changes the epoch number associated with any position $\langle l, in\rangle$ in the received ins by using the values in the received eps (lines 16–19).

Algorithm 4 M^2PAXOS: *Acquisition Phase* (node p_i).

```
1: function Void ACQUISITIONPHASE(Cmd c)
2:     Set ins ← {⟨l, LastDecided[l] + 1⟩ : l ∈ c.LS ∧ ∄in : Decided[l][in] = c}
3:     Array eps
4:     ∀⟨l, in⟩ ∈ ins, eps[l][in] ← + + Epoch[l]
5:     send PREPARE(⟨ins, eps⟩) to all p_k ∈ Π
6:     Set replies ← receive ACKPREPARE(⟨ins, eps, −, −⟩) from Quorum
7:     if ∃⟨ins, eps, NACK, −⟩ ∈ replies then
8:         trigger C-PROPOSE(c)
9:     else
10:        Cmd toForce ← SELECT(ins, replies)
11:        Bool r ← ACCEPTPHASE(toForce, c, ins, eps)
12:        if r = ⊥ ∨ (∃l, in : toForce[l][in] = ⟨v, r⟩ ∧ v ≠ c) then
13:            trigger C-PROPOSE(c)
14:
15: upon PREPARE(⟨Set ins, Array eps⟩) from p_j
16:    if ∀⟨l, in⟩ ∈ ins, Rnd[l][in] < eps[l][in] then
17:        ∀⟨l, in⟩ ∈ ins, Rnd[l][in] ← eps[l][in]
18:        Set decs ← {⟨l, in, Vdec[l][in], Rdec[l][in]⟩ : ⟨l, in⟩ ∈ ins}
19:        send ACKPREPARE(⟨ins, eps, ACK, decs⟩) to p_j
20:    else
21:        send ACKPREPARE(⟨ins, eps, NACK, decs⟩) to p_j
22: function Set SELECT(Set ins, Set replies)
23:    Array toForce
24:    for all ⟨l, in⟩ ∈ ins do
25:        Epoch k ← max({r : ⟨l, in, −, r⟩ ∈ decs ∧ ⟨−, −, −, decs⟩ ∈ replies})
26:        Cmd r ← v : ⟨l, in, v, k⟩ ∈ decs ∧ ⟨−, −, −, decs⟩ ∈ replies
27:        toForce[l][in] ← ⟨r, k⟩
28:    return toForce
```

The meaning of the last two operations is straightforward. A node acknowledges a PREPARE on a position $\langle l, in\rangle$ by promising that it will never positively reply to any other message for $\langle l, in\rangle$ associated with an epoch number not greater than $eps[l][in]$. In addition, it will force the sender of the PREPARE to take into account any possible previous command already issued by another proposal and possibly accepted in $\langle l, in\rangle$. This step is necessary to guarantee *Consistency* also in scenarios where a node that is in the process of executing an *Accept phase* either crashes or is suspected as crashed.

Afterwards, if p_i receives a quorum of replies without any message marked as NACK (lines 9–11), it can enter the *Accept phase* for c. At this time, the input of that phase also includes the set of commands suggested by the received ACKPREPARE messages. In particular, unlike the *Coordination phase*, in this phase p_i passes the multidimensional array $toForce$ to the *Accept phase*, where $toForce[l][in]$, if not $NULL$, stores the command to be accepted in position $\langle l, in\rangle$ and its epoch number (line 11).

An entry $\langle l, in\rangle$ of the array $toForce$ is computed by the SELECT function as follows (lines 10 and 22–28): $toForce[l][in]$ is equal to $\langle r, k\rangle$ where k is the maximum epoch number suggested by a received ACKPREPARE message and associated with the pair $\langle l, in\rangle$, while r is the command (if any) associated with the epoch number k in the received ACKPREPARE messages. Since the prepare phase is a Paxos prepare phase extended to the case of multiple objects, we can inherit the Paxos's property such that if the set of commands associated with k is not empty, it contains only one command.

Finally, if the *Accept phase* does not succeed (for the same reasons described in Section V-B2) or p_i did not succeed to decide c on all the objects in $c.LS$ (because $toForce$ was not empty), p_i triggers a new *Coordination phase* by calling C-PROPOSE(c) (lines 12–13).

C. Correctness

In this section we show why M^2PAXOS implements correctly the Generalized Consensus specification, as defined in Section III.

First, if we consider that in M^2PAXOS nodes only decide the content of $Cstructs$ variables (lines 13-14 of Algorithm 3), then the *Non-triviality* property is guaranteed because a node only appends proposed commands in $Cstructs$ (line 13 of Algorithm 3), and *Stability* is guaranteed because $Cstructs$ variables grow monotonically on each node.

We prove that M^2PAXOS guarantees the *Consistency* property by relying on the correctness of Paxos [1]. In particular, we show that: *(A)* M^2PAXOS decides at most one command for each pair of object l and position in, meaning that the value of $Decided[l][in]$ (Algorithm 3), if different from $NULL$, is the same on all nodes; *(B)* a node orders commands in the same way for all the common objects that the commands access; and *(C)* commands that access an object l are appended in $CStructs$ by following the order defined by the elements of the row $Decided[l]$.

We prove *(A)* by showing that the process of deciding a command for a pair $\langle l, in\rangle$ can be actually mapped to the execution of a Paxos instance that is identified by $\langle l, in\rangle$. Let us define the mapping by considering the case in which a node p_i that proposes a command c is not the owner of all the objects accessed by c (lines 16–17 of Algorithm 1). Note that this case is the most complex one because a node has to become the owner of an object before executing an *Accept* phase on that object; the remaining case, where there already exists an owner of that object, is explained later.

In the former case: *i)* p_i picks a new epoch number $eps[l][in]$ for the object l (line 4 of Algorithm 4), and starts *Phase 1a* of a Paxos instance identified by the pair $\langle l, in\rangle$ (lines 4–5 of Algorithm 4), by proposing c via the broadcast of a PREPARE message; *ii)* a node p_j that receives a PREPARE message from p_i for the triple $\langle l, in, eps[l][in]\rangle$, executes *Phase 1b* of the Paxos instance identified by the pair $\langle l, in\rangle$ (lines 15–21 of Algorithm 4), by sending its reply back to p_i via an ACKPREPARE message; *iii)* like in *Phase 2a* of Paxos, p_i waits for a quorum of ACKPREPARE messages with ACK, each one containing the last command accepted by the sender for the instance $\langle l, in\rangle$, and the greatest epoch in which the sender has accepted a command for the instance $\langle l, in\rangle$ (line 6 of Algorithm 4); *iv)* p_i computes the SELECT function on the quorum of received replies by following the picking rule of *Phase 2a* of Paxos applied to the instance $\langle l, in\rangle$ (lines 10, 22–28 of Algorithm 4), and then it broadcasts via an ACCEPT message either the chosen command, if any, or c otherwise (line 11 of Algorithm 4, and lines 2–8 of Algorithm 2); *v)* a node p_j receiving an ACCEPT message with a command c' and an epoch $eps[l][in]$ executes *Phase 2b* of Paxos applied to the instance $\langle l, in\rangle$, and if that is successful, it broadcasts an ACKACCEPT message with ACK, c', and $eps[l][in]$ to all (lines 16–22 of Algorithm 2); *vi)* a node can decide a command c' in position in for an object l if it receives a quorum of ACKACCEPT messages with ACK for c' in the instance $\langle l, in\rangle$, like the learning policy of Paxos (lines 1–10 of Algorithm 3, and lines 9–14 of Algorithm 2).

In the latter case, namely where there already exists a unique owner of all the objects accessed by a proposed command c, M^2PAXOS acts as Multi-Paxos for all the objects $l \in c.LS$ because the node that already has the ownership of l behaves as the designated leader for the instance of Multi-Paxos identified by l. Hence a node p_i that proposes a command c sends c to the current owner of the objects in $c.LS$ (possibly p_i itself, lines 5–15 of Algorithm 1), which runs M^2PAXOS by starting from step *iv)* defined above.

As a result, by relying on the correctness of Paxos, which prevents two nodes from deciding different values, M^2PAXOS guarantees that for each object l and position in, if a node p_i decided a command c in position in for l, and a node p_j decided a command c' in position in for l (i.e., $Decided[l][in] = c$ on p_i, and $Decided[l][in] = c'$ on p_i), then $c = c'$.

Now we prove *(B)*, which means that: given a node p_i, for any two commands c and c', two objects l_1 and l_2, and four positions h, w, k, z, if $Decided[l_1][h] = c$, $Decided[l_1][w] = c'$, $Decided[l_2][k] = c$ and $Decided[l_2][z] = c'$, we have that $h < w$ iff $k < z$. The proof proceeds as follows. When p_i sets $Decided[l_1][h] = c$, it also sets $Decided[l_2][k] = c$, since the same quorum of ACKACCEPT messages accepts c with ACK in both positions h for l_1 and k for l_2 (lines 6–10 of Algorithm 3). Note that is true because, given a command c, a node sends an ACKACCEPT message with ACK for c only if it can accept c for all the objects in $c.LS$ (lines 16–22 of Algorithm 2). Now, if $Decided[l_1][w] = c'$ and $h < w$ by hypothesis, it must be that when c' was proposed for acceptance in position w for l_1, there was already the command c decided in position h for l_1, since M^2PAXOS always chooses for the acceptance on an object a position greater than the last one decided for that object (line 2 of Algorithms 1 and 4). In addition, we just proved that, when M^2PAXOS performs $Decided[l_1][h] = c$, the command c is also decided in position k for l_2, i.e., $Decided[l_2][k] = c$, and therefore it must be that when c' was proposed for acceptance in position w for l_1 and z for l_2, z was at least $k + 1$, and hence $k < z$ (line 2 of Algorithms 1 and 4).

Proving *(C)* is straightforward if we consider that a command c is appended to the $CStructs$ of a node only if, for each object l accessed by c, there exists a position in such that $Decided[l][in] = c$, and for any position $in' < in$, $Decided[l][in'] \neq NULL$ on that node (see lines 12–13 of Algorithm 3).

By *(A)* and *(B)*, given two conflicting commands c and c' in the $CStructs$ of two nodes p_i and p_j, we have that: for each object l that is commonly accessed by c and c', there exist some k and z, where $k < z$, and $Decided[l][k] = c$ and $Decided[l][z] = c'$ on both p_i and p_j. Further, by *(C)*, command c' cannot be appended to any $CStructs$ before c has been appended. Therefore the conflicting commands c and c' are in the same order in the $CStructs$ delivered by both p_i and p_j, hence guaranteeing *Consistency*.

The *Liveness* property, as defined in Section III, is guaranteed under the same assumptions of Paxos, such that at most $f = \lfloor \frac{N}{2} \rfloor$ nodes can be faulty at any time, and a leader election is eventually possible. Indeed, in that case, if a command c has been proposed by a correct node p_i, eventually, if there is no other concurrent and conflicting command with c in the system, p_i succeeds the execution of all the phases of the protocol for c, since no other node attempts to become the owner of any of the objects in $c.LS$, and there always exists a quorum of nodes that acknowledge for messages.

VI. Evaluation Study

We implemented M^2PAXOS and all competitors within a unified framework, written in the Go programming language [29], version 1.4rc1. Go is compiled, garbage collected, and it has built-in support for managing concurrency.

We evaluated M^2PAXOS by comparing it against three other consensus algorithms: EPaxos, Generalized Paxos and Multi-Paxos. We used up to 49 nodes on the Amazon EC2 infrastructure. Unless otherwise stated, each node is a c3.4xlarge instance (Intel Xeon 2.8GHz, 16 cores, 30GB RAM) running Amazon Linux 2014.09.1. All nodes were deployed under a single placement group. Network bandwidth was measured in excess of 7900mbps. Throughout the evaluation, we refer to a local command from a node p_i as a command that operates on objects whose ownership is already held by p_i.

To properly load the system, we injected commands into an open-loop using up to 64 client threads at each node. Commands are accompanied by a 16-byte payload. After issuing each command, a client thread goes to sleep for a configurable amount of time, i.e., think time. To prevent overloading the system, we limit the number of commands still in-flight. The limit is configured for best performance under each deployment, and when it is reached, a node will skip issuing new commands. Except for the experiments in Figure 2, network messages are batched in order to optimize the network utilization. Each datapoint represents the average of at least 5 measurements.

As benchmarks, we implemented a synthetic application that we customized in order to cover different workloads, which span from the most favorable ones (i.e., partitionable with no inter-node conflicts) to those that require command forwarding (i.e., when the accessed objects share a single remote owner), and to those adverse (i.e., when ownership must be acquired from multiple nodes). In addition, we also ported TPC-C [18], the well-known benchmark widely used in on-line transaction processing systems. Our implementation of TPC-C generates commands that are composed of all the parameters needed for executing TPC-C transactions according to the stored procedure model [30], [4] (e.g., the Id of the accessed `warehouse`, the Id of the accessed `district`). However M^2PAXOS is a consensus layer, thus the actual transaction processing has been omitted. The main purpose of evaluating TPC-C is to show the performance of M^2PAXOS when relevant workload characteristics, such as conflict degree and the number of accessed objects, are set by a well-known benchmark.

In the following plots, we do not explicitly report performance measurements when nodes crash. This is because that scenario would be equivalent of migrating the ownerships acquired by the crashed node to the other requesting nodes.

A. Synthetic benchmark

We first evaluated M^2PAXOS under its most favorable conditions. More specifically, all commands touch a single object, and a command proposed by a node can only conflict with commands proposed by the same node. This scenario is representative for partitioned objects, where replication is only employed for fault-tolerance.

We evaluated the scalability of each consensus protocol as we scaled the system up from 3 to 49 nodes. Figure 1 shows the maximum throughput achieved. In other words, for each configuration tested, we loaded the system up to its saturation and we collected the throughput right before reaching that point. M^2PAXOS provides a significant improvement (i.e., up to 3-7×) when compared to the nearest competitor (i.e., EPaxos), it exhibits great scalability until 11 nodes, and its throughput keeps increasing past 11 nodes, albeit at a slower rate. Multi-Paxos is a distant runner-up at 11 nodes and below, and its performance degrades due to the single leader saturating its computational resources (which are mainly the CPU utilization and the network socket management). After that, it leaves the way for EPaxos, which almost manages to maintain its throughput up to 49 nodes.

Figure 2 shows the median command latency with a system without batching network messages. This way, it is clear the end-to-end latency per command that is experienced by the application. With a low number of nodes, M^2PAXOS narrowly wins over Multi-Paxos, having its latency lower by 23%. As the number of nodes is increased, M^2PAXOS remains the fastest to deliver, with up to 41% better latency than EPaxos.

In practice however, a system is not always maintained at full capacity. Therefore we also explored a more practical deployment with a fixed client workload at each node, in order to assess the scalability of our proposal. Figure 3 reports the throughput of all competitors when the number of clients per node is kept fixed while the node count increases. This way we assess how M^2PAXOS scales up its performance. The results show that, unlike the others, M^2PAXOS exhibits near-linear scalability because it does not generate high contention at the network layer.

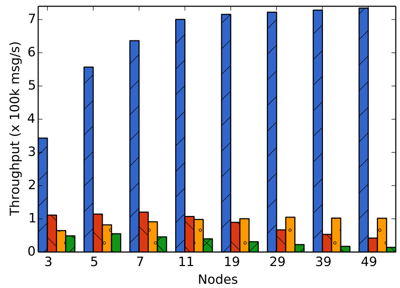


Fig. 1. Maximum attainable throughput varying the number of nodes. Command locality is 100%.

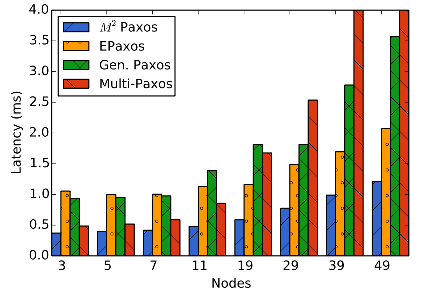


Fig. 2. Median latency without batching network messages. Command locality is 100%.

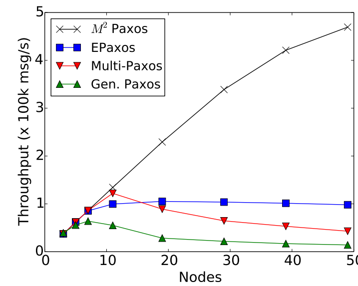
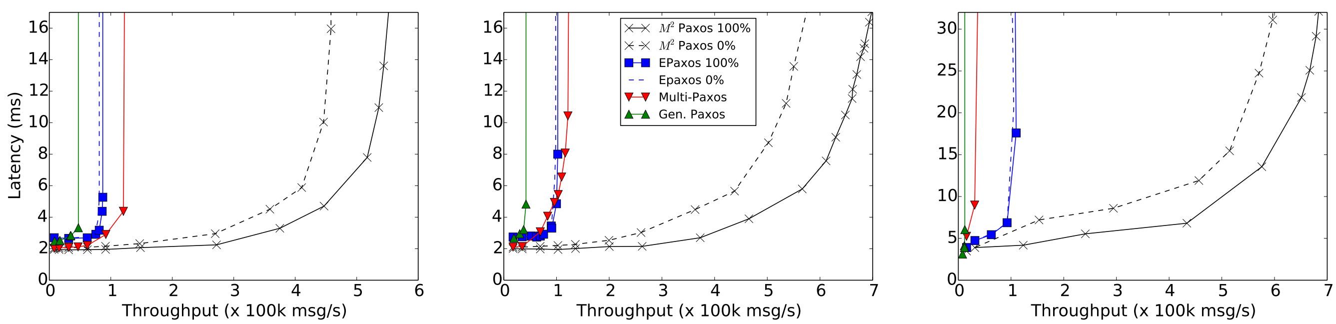


Fig. 3. Scalability. 64 client threads per node, and 5 ms think time. Command locality is 100%.



(a) 5 nodes

(b) 11 nodes

(c) 49 nodes

Fig. 5. Latency vs. throughput plots, with 0% and 100% command locality for M^2PAXOS and EPaxos.

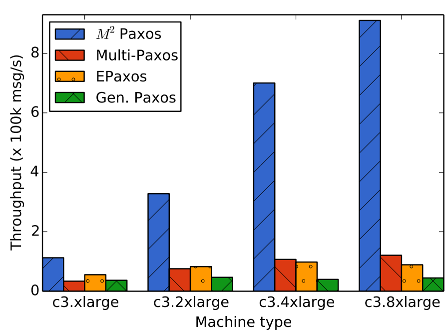


Fig. 4. Maximum throughput for 11-nodes deployments with different machine types. The number of cores are 4, 8, 16 and 32 respectively.

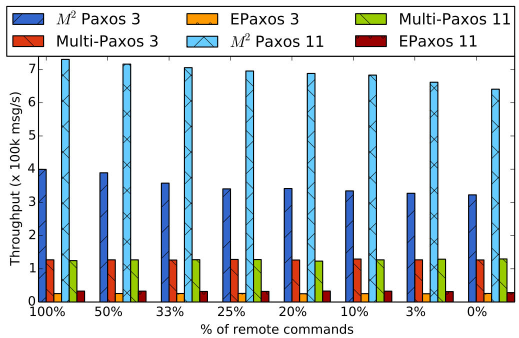
Summarizing, by the analysis of Figures 1, 2, and 3 we can point out weaknesses of the other competitors, which are overcome by M^2PAXOS. On the one hand, both Generalized Paxos and Multi-Paxos suffer from the single leader design, which prevents performance from scaling when the size of the deployment increases. On the other hand, although EPaxos allows multiple leaders to concurrently establish the order of an issued command without contacting a single designated node, its characteristics hamper the achievement of high performance when the number of nodes goes beyond 7.

In fact, EPaxos requires a bigger size of quorum in order to deliver a command in two communication delays in configurations with more than 5 nodes, unlike M^2PAXOS. As a result, as showed in Figure 3, EPaxos provides performance similar to M^2PAXOS up to 7 nodes, where the size of M^2PAXOS's quorum and EPaxos's quorum is comparable. After that, the gap in performance becomes substantial. In addition to that, EPaxos requires the identification of dependent commands in order to deliver *fast*. The meta-data are shared between local threads, thus introducing contention that can lead to poor CPU utilization (an evidence of that is in Figure 4). The overhead of maintaining dependency relations kicks in also when commands are sent through the network because dependencies should be included in the messages themselves. As a consequence of that, messages are bigger and thus they require more time to traverse the network links.

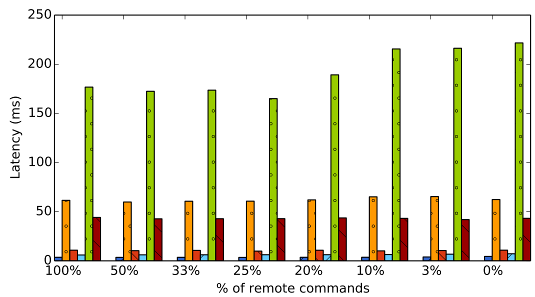
We further evaluated how consensus protocols scale when the number of nodes in a deployment is held constant, and the CPU capacity of each node is increased from 4 to 32 cores. This is relevant for the implementations of Generalized Consensus (which include EPaxos) in order to assess their ability to exploit parallelism in case of low or no conflicts among commands. To this purpose, we ran our benchmark on four classes of Amazon EC2 machines. Each class increment represents a doubling of the number of CPU cores, and an almost 2× increase in available RAM.

Figure 4 shows the result of this experiment on four deployments of 11 nodes each. M^2PAXOS exhibits great scalability up to 16 cores. Throughput still increases beyond that, but at a lower rate, as other components of the system become bottlenecked (more specifically, the networking layer). Clearly this scalability is not exploited by single leader algorithms. Also, EPaxos is not able to take advantage of the additional local resources available because of the cost of dependency management and graph processing, both of which require synchronization among local threads. M^2PAXOS does not require any local processing that generates contention among threads, therefore having more CPUs increases also the parallel tasks accomplished per time unit.

Then, we evaluate the behavior of M^2PAXOS for workloads that exhibit some level of inter-node conflict (Figures 5 and 6) and commands accessing one object. To do that, we show two sets of experiments varying the percentage of local commands.



(a) Throughput



(b) Latency

Fig. 6. Performance varying the probability of proposing a non-local (remote) command. The deployment consists of 3 and 11 nodes.

In Figure 5 we report the latency vs. throughput plots for several deployments (5, 11, and 49 nodes). For M^2PAXOS and EPaxos we plot the results of running two workloads at opposite sides of the locality spectrum where commands still access one object. One workload has perfect locality (100% local commands) and is the best case for M^2PAXOS, where commands proposed by a node only conflict with commands from the same node; the other workload has no locality (0% local commands). Any other workload would fall between these two limits. Multi-Paxos and Generalized Paxos are not sensitive to locality, while M^2PAXOS handles non-local commands by simply forwarding them to the node that currently owns the requested object (see also Section IV-B). In such a scenario, EPaxos can fail in delivering a transaction fast due to the collection of conflicting dependencies during its broadcast phase. For this reason, it breaks down up to 10% earlier in the workload with no locality.

In Figure 6 we show the performance of all competitors given two configurations with 3 and 11 nodes, and by varying the percentage of non-local commands with a finer granularity than that in Figure 5. Here the impact of the forwarding mechanism of M^2PAXOS is evident. The performance degradation is very small (on average 4%), whereas other competitors already achieved their top performance, thus changing the probability of issuing a local command does not provide significant performance improvement or degradation, respectively.

The last tested scenario using the synthetic benchmark is where commands are complex. We define complex commands as the commands that access multiple objects, hence potentially conflicting with commands from multiple nodes. Specifically, in this experiment a complex command accesses one object in a set, called *local-set*, on which the local node is likely to have the ownership, and one uniformly distributed across all objects. In this configuration, we fixed the number of nodes as 49 and we varied the size of local-set. The results (in Figure 7) show a drop in throughput as the fraction of complex commands is increased. The drop rate and final throughput all depend on the the size of local-set because it affects the contention rate. Multi-Paxos and Generalized Paxos are not affected by the presence of complex commands. EPaxos exhibits a small reduction in throughput as the percentage of complex commands nears 100%. However, M^2PAXOS is able to sustain the throughput by even using almost 50% of complex commands, in case the size of local-set is 1000.

One important observation, which is valid for both EPaxos and Generalized Paxos, is that when complex commands are deployed, messages on the network become much bigger due to the presence of dependency relations to include. It is worth mentioning that protocols like EPaxos have to also include dependencies from other local threads that may issue a conflicting command. M^2PAXOS does not suffer from such a drawback because it does not rely on command dependencies and local threads can proceed in parallel as long as the node has the ownership on those objects.

B. TPC-C benchmark

In this evaluation study we included also a benchmark that produces the same workload as TPC-C. We configured it by deploying a total number of `warehouses` equal to 10*N (e.g., with 9 nodes we deployed 90 `warehouses`). Following the benchmark specification, we associated the appropriate number of `customers`, `districts`, `items`, etc. TPC-C has five transaction profiles, where each of them has a set of indexes identifying the objects to access (e.g., the `warehouse` Id). Those indexes corresponds to the payload of the complex command we issue. We define a warehouse to be local to a node if its `warehouse` object and all the objects related with it (e.g., its `districts`) belong to the local-set of that node.

Figure 8 shows the performance by varying the likelihood for a thread to broadcast a command on a local `warehouse` (Figure 8(a)), rather than on a `warehouse` (Figure 8(b)) uniformly selected across all. According to the specification of TPC-C, even though the requested `warehouse` is the local one, 15% of the *payment* transactions (a profile of TPC-C) can still access a `customer` belonging to another `warehouse`.

We first notice that, the overall throughput provided by M^2PAXOS is less than the one obtained before with the single-object command cases. This is because TPC-C transaction profiles need more than 3 parameters to execute, thus, accordingly, commands' size is bigger. Performance decreases further (by as much as 40%) when we let the benchmark

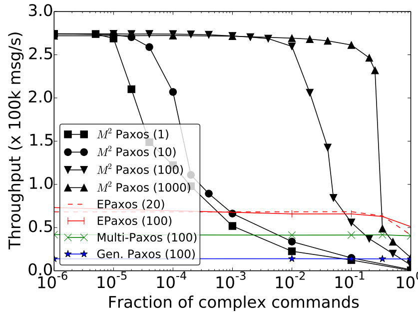
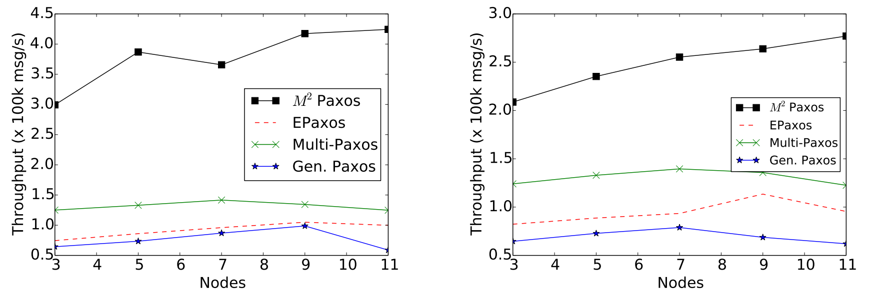


Fig. 7. Throughput varying the fraction of complex commands with 49 nodes. In parentheses the number of possible objects per node.



(a) 0% of commands on a remote warehouse

(b) 15% of commands on a remote warehouse

Fig. 8. Performance with TPC-C workload by varying the number of nodes up to 11.

access a non-local `warehouse` for the 15% of the cases. However, still M^2PAXOS provides a throughput greater than 400k commands ordered per second in the configuration of Figure 8(a), and more than 250k under the configuration of Figure 8(b).

The closest competitor (but still 2.4× slower) is Multi-Paxos. The reason is related to the difficulties experienced by EPaxos (5.5× slower) on handling higher contention, which leads the agreement phase to perform an additional ordering phase after trying (and failing) to deliver fast. Multi-Paxos's performance is independent from the message composition and the overall application contention because the total order it produces does not take into account any conflict among messages. In fact, it performs similar to the results in Figure 6(a).

VII. CONCLUSION

In this paper we presented M^2PAXOS, an algorithm providing a scalable and high-performance implementation of Generalized Consensus. It is able to decide sequences of commands with the optimal cost of two communication delays in the case of partitionable workload and with the minimum size of quorums achievable for solving consensus in asynchronous systems, i.e., $\lfloor \frac{N}{2} \rfloor + 1$, where N is the total number of nodes. The evaluation study of M^2PAXOS confirms the effectiveness of the approach by gaining as much as 7× over state-of-the-art consensus and generalized consensus algorithms.

VIII. ACKNOWLEDGMENTS

This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-15-1-0098 and by US National Science Foundation under grant CNS-1523558.

REFERENCES

[1] L. Lamport, "The Part-time Parliament," *ACM TOCS*, 1998.

[2] B. Charron-Bost and A. Schiper, "Uniform Consensus is Harder Than Consensus," *J. Algorithms*, vol. 51, no. 1, pp. 15–37, Apr. 2004.

[3] J. C. Corbett *et al.*, "Spanner: Google's Globally Distributed Database," *ACM TOCS*, 2013.

[4] S. Hirve, R. Palmieri, and B. Ravindran, "Archie: a speculative replicated transactional system," in *Middleware*, 2014, pp. 265–276.

[5] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data Center Consistency," in *EuroSys 2013*.

[6] H. Mahmoud *et al.*, "Low-latency Multi-datacenter Databases Using Replicated Commit," *Proc. VLDB Endow.*, 2013.

[7] L. Lamport, "Paxos made simple," *ACM Sigact News*, 2001.

[8] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is More Consensus in Egalitarian Parliaments," in *SOSP 2013*.

[9] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs," in *OSDI 2008*.

[10] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, "Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems," in *OPODIS 2014*.

[11] L. Lamport, "Generalized Consensus and Paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005.

[12] F. Pedone and A. Schiper, "Generic Broadcast," in *DISC*, 1999, pp. 94–108.

[13] L. Lamport, "Fast paxos," *Distributed Computing*, 2006.

[14] ——, "Future directions in distributed computing." Springer-Verlag, 2003, ch. Lower Bounds for Asynchronous Consensus.

[15] J. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," ser. USENIX ATC, 2012, pp. 21–21.

[16] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, 2012, pp. 1–12.

[17] S. Peluso, P. Romano, and F. Quaglia, "SCORe: A Scalable One-copy Serializable Partial Replication Protocol," in *Middleware 2012*.

[18] "Tpc-c benchmark," http://www.tpc.org/tpcc/.

[19] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[20] P. Sutra and M. Shapiro, "Fast genuine generalized consensus," in *SRDS 2011*.

[21] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous Lease-based Replication of Software Transactional Memory," in *Middleware 2010*.

[22] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, and A. Suissa, "Exploiting Locality in Lease-Based Replicated Transactional Memory via Task Migration," in *DISC 2013*.

[23] M. J. Fischer *et al.*, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, 1985.

[24] R. Guerraoui and A. Schiper, "Genuine Atomic Multicast in Asynchronous Distributed Systems," *Elsevier TCS 2001*.

[25] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., 2006.

[26] B. Charron-Bost and A. Schiper, "Improving fast paxos: being optimistic with no overhead," in *PRDC 2006*.

[27] F. Junqueira, Y. Mao, and K. Marzullo, "Classic paxos vs. fast paxos: Caveat emptor," in *HotDep*, 2007.

[28] R. Guerraoui, V. Kuncak, and G. Losa, "Speculative linearizability," ser. PLDI, 2012, pp. 55–66.

[29] "The go programming language." http://golang.org/.

[30] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1018–1032, 2003.

Appendix

MODULE *MultiConsensus*

A set of constants and definitions for use in the specification of MultiPaxos-like algorithms.

$\text{EXTENDS } \mathit{Integers}, \mathit{FiniteSets}$

$\text{CONSTANTS } \mathit{Acceptors}, \mathit{Quorums}, V, \mathit{None}$

$\text{ASSUME } \mathit{None} \notin V$

$\text{ASSUME } \forall Q \in \mathit{Quorums} : Q \subseteq \mathit{Acceptors}$

$\text{ASSUME } \forall Q1, Q2 \in \mathit{Quorums} : Q1 \cap Q2 \neq \{\}$

$\mathit{Ballots} \triangleq \mathit{Nat}$

$\text{ASSUME } -1 \notin \mathit{Ballots}$

$\mathit{Instances} \triangleq \mathit{Nat}$

$\mathit{MajQuorums} \triangleq \{Q \in \text{SUBSET } \mathit{Acceptors} : \mathit{Cardinality}(Q) > \mathit{Cardinality}(\mathit{Acceptors}) \div 2\}$

$\mathit{Max}(xs, \mathit{LessEq}(\_, \_)) \triangleq \text{CHOOSE } x \in xs : \forall y \in xs : \mathit{LessEq}(y, x)$

MODULE *MultiPaxos*

An abstract specification of the *MultiPaxos* algorithm. We do not model the network nor leaders explicitely. Instead, we keep the history of all votes cast and use this history to describe how new votes are cast. Note that, in some way, receiving a message corresponds to reading a past state of the sender. We produce the effect of having the leader by requiring that not two different values can be voted for in the same ballot.

This specification is inspired from the abstract specification of Generalized *Paxos* presented in the Generalized *Paxos* paper by *Lamport*.

EXTENDS *MultiConsensus*

The variable ballot maps an acceptor to its current ballot.

Given an acceptor a, an instance i, and a ballot b, $vote[a][i][b]$ records the vote that a casted in ballot b of instance i.

VARIABLES

$ballot, vote, propCmds$

$$
\begin{aligned}
&Init \triangleq\\
&\quad \land\ ballot = [a \in Acceptors \mapsto -1]\\
&\quad \land\ vote = [a \in Acceptors \mapsto\\
&\qquad\qquad [i \in Instances \mapsto\\
&\qquad\qquad\quad [b \in Ballots \mapsto None]]]\\
&\quad \land\ propCmds = \{\}
\end{aligned}
$$

$$
\begin{aligned}
&TypeInv \triangleq\\
&\quad \land\ ballot \in [Acceptors \to \{-1\} \cup Ballots]\\
&\quad \land\ vote \in [Acceptors \to\\
&\qquad\qquad [Instances \to\\
&\qquad\qquad\quad [Ballots \to \{None\} \cup V]]]\\
&\quad \land\ propCmds \in \text{SUBSET } V
\end{aligned}
$$

Properties of ballot and vote

The maximal ballot in which an acceptor a voted is always less than or equal to its current ballot.

$$
\begin{aligned}
&WellFormed \triangleq \forall a \in Acceptors : \forall i \in Instances : \forall b \in Ballots :\\
&\quad b > ballot[a] \Rightarrow vote[a][i][b] = None
\end{aligned}
$$

$$
\begin{aligned}
&ChosenAt(i, b, v) \triangleq\\
&\quad \exists Q \in Quorums : \forall a \in Q : vote[a][i][b] = v
\end{aligned}
$$

$$
\begin{aligned}
&Chosen(i, v) \triangleq\\
&\quad \exists b \in Ballots : ChosenAt(i, b, v)
\end{aligned}
$$

$$
\begin{aligned}
&Choosable(v, i, b) \triangleq\\
&\quad \exists Q \in Quorums : \forall a \in Q : ballot[a] > b \Rightarrow vote[a][i][b] = v
\end{aligned}
$$

$SafeAt(v, i, b) \triangleq$

$\forall b2 \in Ballots : \forall v2 \in V :$
$\quad (b2 < b \land Choosable(v2, i, b2))$
$\quad \Rightarrow v = v2$

$SafeInstanceVoteArray(i) \triangleq \forall b \in Ballots : \forall a \in Acceptors :$
$\quad \text{LET } v \triangleq vote[a][i][b]$
$\quad \text{IN } v \neq None \Rightarrow SafeAt(v, i, b)$

$SafeVoteArray \triangleq \forall i \in Instances : SafeInstanceVoteArray(i)$

If the vote array is well formed and the vote array is safe, then for each instance only a unique value can be chosen.

$\text{THEOREM } TypeInv \land WellFormed \land SafeVoteArray \Rightarrow \forall i \in Instances :$
$\quad \forall v1, v2 \in V : Chosen(i, v1) \land Chosen(i, v2) \Rightarrow v1 = v2$

A ballot is conservative when all acceptors which vote in the ballot vote for the same value. In *MultiPaxos*, the leader of a ballot ensures that the ballot is conservative.

$Conservative(i, b) \triangleq$
$\quad \forall a1, a2 \in Acceptors :$
$\quad\quad \text{LET } v1 \triangleq vote[a1][i][b]$
$\quad\quad\quad v2 \triangleq vote[a2][i][b]$
$\quad\quad \text{IN } (v1 \neq None \land v2 \neq None) \Rightarrow v1 = v2$

$ConservativeVoteArray \triangleq$
$\quad \forall i \in Instances : \forall b \in Ballots :$
$\quad\quad Conservative(i, b)$

The maximal ballot smaller than *max* in which a has voted in instance i.

$MaxVotedBallot(i, a, max) \triangleq$
$\quad Max(\{b \in Ballots : b \leq max \land vote[a][i][b] \neq None\} \cup \{-1\}, \leq)$

$MaxVotedBallots(i, Q, max) \triangleq \{MaxVotedBallot(i, a, max) : a \in Q\}$

The vote casted in the maximal ballot smaller than *max* by an acceptor of the quorum Q.

$HighestVote(i, max, Q) \triangleq$
$\quad \text{IF } \exists a \in Q : MaxVotedBallot(i, a, max) \neq -1$
$\quad \text{THEN}$
$\quad\quad \text{LET } MaxVoter \triangleq \text{CHOOSE } a \in Q :$
$\quad\quad\quad MaxVotedBallot(i, a, max) = Max(MaxVotedBallots(i, Q, max), \leq)$
$\quad\quad \text{IN } vote[MaxVoter][i][MaxVotedBallot(i, MaxVoter, max)]$
$\quad \text{ELSE}$
$\quad\quad None$

Values that are safe to vote for in ballot b according to a quorum Q whose acceptors have all reached ballot b.

If there is an acceptor in Q that has voted in a ballot less than b, then the only safe value is the value voted for by an acceptor in Q in the highest ballot less than b.

Else, all values are safe.

In an implementation, the leader of a ballot b can compute $ProvedSafeAt(i, Q, b)$ when it receives $1b$ messages from the quorum Q.

$$
\begin{array}{l}
ProvedSafeAt(i, Q, b) \triangleq \\
\quad \text{IF } HighestVote(i, b-1, Q) \neq None \\
\quad\quad \text{THEN } \{HighestVote(i, b-1, Q)\} \\
\quad\quad \text{ELSE } V
\end{array}
$$

In a well-formed, safe, and conservative vote array, all values that are proved safe are safe.

$$
\begin{array}{l}
\text{THEOREM } TypeInv \land WellFormed \land SafeVoteArray \land ConservativeVoteArray \\
\quad \Rightarrow \quad \forall v \in V : \forall i \in Instances : \\
\quad\quad\quad \forall Q \in Quorums : \forall b \in Ballots : \\
\quad\quad\quad\quad \land\ \forall a \in Q : ballot[a] \geq b \\
\quad\quad\quad\quad \land\ v \in ProvedSafeAt(i, Q, b) \\
\quad\quad\quad\quad \Rightarrow SafeAt(v, i, b)
\end{array}
$$

The propose action:

$$
\begin{array}{l}
Propose(v) \triangleq \\
\quad \land\ propCmds' = propCmds \cup \{v\} \\
\quad \land\ \text{UNCHANGED } \langle ballot, vote \rangle
\end{array}
$$

The *JoinBallot* action: an acceptor can join a higher ballot at any time. In an implementation, the *JoinBallot* action is triggered by a $1a$ message from the leader of the new ballot.

$$
\begin{array}{l}
JoinBallot(a, b) \triangleq \\
\quad \land\ ballot[a] < b \\
\quad \land\ ballot' = [ballot \text{ EXCEPT } ![a] = b] \\
\quad \land\ \text{UNCHANGED } \langle vote, propCmds \rangle
\end{array}
$$

The *Vote* action: an acceptor casts a vote in instance i. This action is enabled when the acceptor has joined a ballot, has not voted in its current ballot, and can determine, by reading the last vote cast by each acceptor in a quorum, which value is safe to vote for. If multiple values are safe to vote for, we ensure that only one can be voted for by requiring that the ballot remain conservative.

In an implementation, the computation of safe values is done by the leader of the ballot when it receives $1b$ messages from a quorum of acceptors. The leader then picks a unique value among the safe values and suggests it to the acceptors.

$$
\begin{array}{l}
Vote(a, v, i) \triangleq \\
\quad \land\ ballot[a] \neq -1 \\
\quad \land\ vote[a][i][ballot[a]] \in \{None, v\} \\
\quad \land\ \exists Q \in Quorums : \\
\quad\quad\quad \land\ \forall q \in Q : ballot[q] \geq ballot[a] \\
\quad\quad\quad \land\ v \in ProvedSafeAt(i, Q, ballot[a]) \cap propCmds \\
\quad\quad\quad \land\ vote' = [vote \text{ EXCEPT } ![a] = \\
\quad\quad\quad\quad\quad [@ \text{ EXCEPT } ![i] = [@ \text{ EXCEPT } ![ballot[a]] = v]]] \\
\quad \land\ \text{UNCHANGED } \langle ballot, propCmds \rangle
\end{array}
$$

$\land \; Conservative(i, \, ballot[a])'$

$Next \triangleq$
$\quad \lor \; \exists \, v \in V : Propose(v)$
$\quad \lor \; \exists \, a \in Acceptors : \exists \, b \in Ballots : JoinBallot(a, \, b)$
$\quad \lor \; \exists \, a \in Acceptors, \, i \in Instances, \, v \in propCmds :$
$\quad\quad Vote(a, \, v, \, i)$

$Correctness \triangleq$
$\quad \forall \, i \in Instances : \forall \, v1, \, v2 \in V :$
$\quad\quad Chosen(i, \, v1) \land Chosen(i, \, v2) \Rightarrow v1 = v2$

$Spec \triangleq Init \land \quad \Box[Next]_{\langle ballot, \, vote, \, propCmds \rangle}$

THEOREM $Spec \Rightarrow \Box Correctness$

MODULE *Objects*

CONSTANTS $Commands$, $AccessedBy(\_)$, $Objects$

$AccessedBy(c)$ is the set of objects accessed by c.

ASSUME $\forall\, c \in Commands : AccessedBy(c) \in \text{SUBSET } Objects$

MODULE *GFPaxos*

An abstract specification of *GFPaxos*. It consists in coordinating several *MultiPaxos* instances (one per object).

EXTENDS $MultiConsensus, Sequences, Objects$

ASSUME $Instances \subseteq Nat \setminus \{0\}$

ASSUME $Commands = V$

ballot and vote are functions from object to "ballot" and "vote" structures of the *MultiPaxos* specification.

VARIABLES
$ballots, votes, propCmds$

The *MultiPaxos* instance of object o.

$MultiPaxos(o) \triangleq$
INSTANCE $MultiPaxos$ WITH
$ballot \leftarrow ballots[o],$
$vote \leftarrow votes[o]$

$InitBallot \triangleq [a \in Acceptors \mapsto -1]$
$InitVote \triangleq [a \in Acceptors \mapsto [i \in Instances \mapsto [b \in Ballots \mapsto None]]]$

The initial state

$Init \triangleq$
$\land\ ballots = [o \in Objects \mapsto InitBallot]$
$\land\ votes = [o \in Objects \mapsto InitVote]$
$\land\ propCmds = \{\}$

Is instance i of object o complete?

$Complete(o, i) \triangleq$
$\exists v \in V : MultiPaxos(o)!Chosen(i, v)$

The next undecided instance for object o:

$NextInstance(o) \triangleq$
LET $completed \triangleq \{i \in Instances : Complete(o, i)\}$
IN IF $completed \neq \{\}$
THEN $Max(completed, \leq) + 1$
ELSE $Max(Instances, \geq)$ the minimum instance

The next-state relation:

Either an acceptor executes the *JoinBallot* action in the *MultiPaxos* instance of an object o, or, for a command c, an acceptor executes the Vote action in all instances that correspond to an object that the command c accesses.

Note that for each object o, an acceptor only votes in the instance whose predecessor is the largest instance in which a command was decided for o, using a non-distributed implementation.

Join a higher ballot for an object:

$$JoinBallot(a, o, b) \triangleq$$
$$\land\ MultiPaxos(o)!JoinBallot(a, b)$$
$$\land\ \forall obj \in Objects \setminus \{o\} : \text{UNCHANGED } \langle ballots[obj], votes[obj] \rangle$$

Vote for c in all of the instances of c's objects:

$$Vote(a, c) \triangleq$$
$$\land \exists is \in [AccessedBy(c) \rightarrow Instances] :$$
$$\quad \land\ \forall obj \in AccessedBy(c) : is[obj] \leq NextInstance(obj)$$
$$\quad \land\ \forall o \in AccessedBy(c) :$$
$$\qquad MultiPaxos(o)!Vote(a, c, is[o])$$
$$\land\ \forall o \in Objects \setminus AccessedBy(c) : \text{UNCHANGED } \langle ballots[o], votes[o] \rangle$$

$$Propose(v) \triangleq$$
$$\land\ propCmds' = propCmds \cup \{v\}$$
$$\land\ \text{UNCHANGED } \langle ballots, votes \rangle$$

$$Next \triangleq$$
$$\lor\ \exists c \in V : Propose(c)$$
$$\lor\ \exists o \in Objects : \exists a \in Acceptors : \exists b \in Ballots :$$
$$\qquad JoinBallot(a, o, b)$$
$$\lor\ \exists c \in Commands : \exists a \in Acceptors :$$
$$\qquad Vote(a, c)$$

$$Spec \triangleq Init \land \Box[Next]_{\langle ballots, votes, propCmds \rangle}$$

Correctness properties.

True when $c1$ has been chosen before $c2$ in the *MultiPaxos* instance associated to object o. This definition works only when there are no duplicate chosen commands.

$$ChosenInOrder2(c1, c2, o) \triangleq$$
$$\land\ c1 \neq c2$$
$$\land\ \exists i, j \in Instances :$$
$$\quad \land\ MultiPaxos(o)!Chosen(i, c1)$$
$$\quad \land\ MultiPaxos(o)!Chosen(j, c2)$$
$$\quad \land\ i < j$$

Have the commands in cs been chosen in instances of object o?

$$Chosen(cs, o) \triangleq$$
$$\forall c \in cs : \exists i \in Instances : MultiPaxos(o)!Chosen(i, c)$$

A simplified correctness property: any two commands are ordered in the same way by the *MultiPaxos* instances corresponding to objects that both commands access. This correctness property is satisfied only if no duplicate commands can be chosen.

$$CorrectnessSimple \triangleq$$
$$\forall c1, c2 \in Commands : \forall o1, o2 \in AccessedBy(c1) \cap AccessedBy(c2) :$$
$$\quad \land ChosenInOrder2(c1, c2, o1)$$
$$\quad \land Chosen(\{c1, c2\}, o2)$$

$\Rightarrow \mathit{ChosenInOrder2}(c1,\ c2,\ o2)$

A more complex correctness condition that is satisfied by the spec, even in the presence of duplicate commands

Removing duplicates from a seqence

$$
\begin{array}{l}
\textsc{recursive}\ \mathit{RemDupRec}(\_,\ \_) \\
\ \mathit{RemDupRec}(es,\ seen)\ \triangleq \\
\quad \textsc{if}\ es = \langle\rangle \\
\quad\ \ \textsc{then}\ \langle\rangle \\
\quad\ \ \textsc{else} \\
\quad\quad \textsc{if}\ es[1] \in seen \\
\quad\quad\ \ \textsc{then}\ \mathit{RemDupRec}(\mathit{Tail}(es),\ seen) \\
\quad\quad\ \ \textsc{else}\ \ \langle es[1]\rangle \circ \mathit{RemDupRec}(\mathit{Tail}(es),\ seen \cup \{es[1]\})
\end{array}
$$

$$\mathit{RemDup}(es)\ \triangleq\ \mathit{RemDupRec}(es,\ \{\})$$

For each object, the sequence of commands chosen with duplicates removed.

$$
\begin{array}{l}
\mathit{ChosenCmds}\ \triangleq\ [o \in \mathit{Objects} \mapsto \\
\quad \textsc{let}\ s\ \triangleq\ [i \in \mathit{Instances} \mapsto \\
\quad\quad\quad\quad\quad\quad \textsc{if}\ \exists\, c \in \mathit{propCmds} : \mathit{MultiPaxos}(o)!\mathit{Chosen}(i,\ c) \\
\quad\quad\quad\quad\quad\quad\ \ \textsc{then}\ \textsc{choose}\ c \in \mathit{propCmds} : \mathit{MultiPaxos}(o)!\mathit{Chosen}(i,\ c) \\
\quad\quad\quad\quad\quad\quad\ \ \textsc{else}\ \ \mathit{None}] \\
\quad \textsc{in}\quad \mathit{RemDup}(s)]
\end{array}
$$

The image of a function.

$$\mathit{Image}(f)\ \triangleq\ \{f[x] : x \in \textsc{domain}\ f\}$$

Has $c1$ been chosen before $c2$ for object o?

$$
\begin{array}{l}
\mathit{ChosenInOrder}(c1,\ c2,\ o)\ \triangleq \\
\quad \textsc{let}\ s\ \triangleq\ \mathit{ChosenCmds}[o] \\
\quad \textsc{in} \\
\quad\quad\quad \land\ \{c1,\ c2\} \subseteq \mathit{Image}(s) \\
\quad\quad\quad \land\ \forall\, i,\ j \in \textsc{domain}\ s : \\
\quad\quad\quad\quad\quad s[i] = c1 \land s[j] = c2 \Rightarrow i \leq j
\end{array}
$$

Correctness: if two commands have been ordered for two different objects, then their order is the same.

$$
\begin{array}{l}
\mathit{Correctness}\ \triangleq\ \ \forall\, c1,\ c2 \in \mathit{Commands} : \\
\quad \forall\, o1,\ o2 \in \mathit{AccessedBy}(c1) \cap \mathit{AccessedBy}(c2) : \\
\quad\quad (\forall\, o \in \{o1,\ o2\} : \\
\quad\quad\quad\quad c1 \in \mathit{Image}(\mathit{ChosenCmds}[o]) \land c2 \in \mathit{Image}(\mathit{ChosenCmds}[o])) \\
\quad\quad\ \Rightarrow\ \ (\mathit{ChosenInOrder}(c1,\ c2,\ o1) = \mathit{ChosenInOrder}(c1,\ c2,\ o2))
\end{array}
$$

$\textsc{theorem}\ \mathit{Spec} \Rightarrow \Box\mathit{Correctness}$

The spec above cannot be used with TLC because TLC does not accept statements like $fun[x]' = y$ (updating the value of a function on just a subset of its domain), and that's what happens when we reuse the specification of *MultiPaxos*. Below is a second version of the spec, which should be equivalent to the one above, and which can be model-checked with TLC.

$$
\begin{array}{l}
JoinBallot2(a,\ o,\ b)\ \triangleq \\
\quad \land\ ballots' = [ballots \text{ EXCEPT } ![o] = [ballots[o] \text{ EXCEPT } ![a] = b]] \\
\quad \land\ \text{UNCHANGED } votes \\
\quad \land\ MultiPaxos(o)!JoinBallot(a,\ b)
\end{array}
$$

$Vote2(c,\ a)\ \triangleq$

Vote for c in all of the instances of c's objects:

$$
\begin{array}{l}
\quad \land \exists\, is \in [AccessedBy(c) \rightarrow Instances] : \\
\qquad \land\ \forall\, obj \in AccessedBy(c) : is[obj] \leq NextInstance(obj) \\
\qquad \land\ votes' = [o \in Objects \mapsto \\
\qquad\qquad \text{IF } o \in AccessedBy(c) \\
\qquad\qquad\ \text{THEN} \\
\qquad\qquad\quad [votes[o] \text{ EXCEPT } ![a] = [@ \text{ EXCEPT } ![is[o]] = \\
\qquad\qquad\qquad \text{IF } ballots[o][a] \neq -1 \\
\qquad\qquad\qquad\ \text{THEN } [@ \text{ EXCEPT } ![ballots[o][a]] = c] \\
\qquad\qquad\qquad\ \text{ELSE } @]] \\
\qquad\qquad\ \text{ELSE } votes[o]] \\
\quad \land\ \text{UNCHANGED } ballots
\end{array}
$$

Only do the updates above if all of the instances can take the transition according to *MultiPaxos*:

$$
\begin{array}{l}
\quad \land\ \forall\, o \in AccessedBy(c) : \exists\, i \in Instances : \\
\qquad MultiPaxos(o)!Vote(a,\ c,\ i)
\end{array}
$$

An equivalent version of *Next* which can be used with TLC

$$
\begin{array}{l}
Next2\ \triangleq \\
\quad \lor\ \exists\, o \in Objects : \exists\, a \in Acceptors : \exists\, b \in Ballots : \\
\qquad JoinBallot2(a,\ o,\ b) \\
\quad \lor\ \exists\, c \in Commands : \exists\, a \in Acceptors : \\
\qquad Vote2(c,\ a) \\
\quad \lor\ \exists\, c \in V : Propose(c)
\end{array}
$$

$$Spec2\ \triangleq\ Init \land \Box[Next2]_{\langle ballots,\ votes,\ propCmds \rangle}$$

Model-checking results:

Model: 3 acceptors, 2 objects, 2 commands (1 accessing both, 1 accessing only 1 object), majority quorums, 3 ballots, 3 instances.

Checked *CorrectnessSimple*.

State constraint to avoid duplicate commands and overflows caused by accessing $votes[a][o][NextInstance(i)]$ when all instances are complete:

$\quad \land \forall\, o \in Objects : \exists\, i \in Instances : \neg Complete(o,\ i)$

$\wedge \ \forall\, o \in \mathit{Objects} : \forall\, a \in \mathit{Acceptors} : \forall\, i \in \mathit{Instances} : \forall\, c \in \mathit{Commands} : \neg \mathit{MultiPaxos}(o)!\mathit{Chosen}(i,\ \mathit{votes}[o][a][i])$

Running on 48 *Xeon* cores with 120*GB* of memory.

Exhaustive exploration completed: 674414109 states generated, 48486426 distinct states found. The depth of the complete state graph search is 31.

MODULE *DistributedMultiPaxos*

A specification of *MultiPaxos* that includes a model of the network. Compared to the abstract specification, processes now communicate through the network instead of directly reading each other's state. The main difference is that network messages reflect a past state of their sender, not its current state. Note that since the state of the processes is monothonic (*i.e.* values written in the vote array are never overwritten and ballots on increase), knowing the past state gives some information about the current state.

EXTENDS *MultiConsensus*

VARIABLES

$\quad ballot, vote, network, propCmds$

We do not model learners, so no need for $2b$ messages

$$
\begin{array}{l}
Msgs \triangleq \\
\quad \{\langle \text{“1a”}, b\rangle : b \in Ballots\} \cup \\
\quad \{\langle \text{“1b”}, a, i, b, \langle maxB, v\rangle\rangle : i \in Instances, a \in Acceptors, \\
\quad\quad b \in Ballots, maxB \in Ballots \cup \{-1\}, v \in V \cup \{None\}\} \cup \\
\quad \{\langle \text{“2a”}, i, b, v\rangle : i \in Instances, b \in Ballots, v \in V\}
\end{array}
$$

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ ballot = [a \in Acceptors \mapsto -1] \\
\quad \land\ vote = [a \in Acceptors \mapsto \\
\qquad\qquad [i \in Instances \mapsto \\
\qquad\qquad\quad [b \in Ballots \mapsto None]]] \\
\quad \land\ network = \{\} \\
\quad \land\ propCmds = \{\}
\end{array}
$$

$$
\begin{array}{l}
TypeInv \triangleq \\
\quad \land\ ballot \in [Acceptors \to \{-1\} \cup Ballots] \\
\quad \land\ vote \in [Acceptors \to \\
\qquad\qquad [Instances \to \\
\qquad\qquad\quad [Ballots \to \{None\} \cup V]]] \\
\quad \land\ network \subseteq Msgs \\
\quad \land\ propCmds \subseteq V
\end{array}
$$

$$
\begin{array}{l}
Propose(c) \triangleq \\
\quad \land\ propCmds' = propCmds \cup \{c\} \\
\quad \land\ \text{UNCHANGED } \langle ballot, vote, network\rangle
\end{array}
$$

$$
\begin{array}{l}
Phase1a(b) \triangleq \\
\quad \land\ network' = network \cup \{\langle \text{“1a”}, b\rangle\} \\
\quad \land\ \text{UNCHANGED } \langle ballot, vote, propCmds\rangle
\end{array}
$$

A pair consisting of the highest ballot in which the acceptor a has voted in instance i. If a has not voted in instance i, then $\langle -1, None\rangle$.

$MaxAcceptorVote(a, i) \triangleq$

$$
\begin{array}{l}
\text{LET } maxBallot \triangleq Max(\{b \in Ballots : vote[a][i][b] \neq None\} \cup \{-1\}, \leq) \\
\quad\; v \triangleq \text{IF } maxBallot > -1 \text{ THEN } vote[a][i][maxBallot] \text{ ELSE } None \\
\text{IN } \langle maxBallot, v \rangle
\end{array}
$$

Acceptor a receives responds from a $1a$ message by sending, for each instance i, its max vote in this instance.

$$
\begin{array}{l}
Phase1b(a, b, v) \triangleq \\
\quad \land\; ballot[a] < b \\
\quad \land\; \langle \text{"1a"}, b \rangle \in network \\
\quad \land\; ballot' = [ballot \text{ EXCEPT } ![a] = b] \\
\quad \land\; network' = network \cup \\
\qquad\quad \{\langle \text{"1b"}, a, i, b, MaxAcceptorVote(a, i) \rangle : i \in Instances\} \\
\quad \land\; \text{UNCHANGED } \langle vote, propCmds \rangle
\end{array}
$$

$$
\begin{array}{l}
1bMsgs(b, i, Q) \triangleq \\
\quad \{m \in network : m[1] = \text{"1b"} \land m[2] \in Q \land m[3] = i \land m[4] = b\}
\end{array}
$$

The vote cast in the highest ballot less than b in instance i. This vote is unique because all ballots are conservative. Note that this can be *None*.

$$
\begin{array}{l}
MaxVote(b, i, Q) \triangleq \\
\quad \text{LET } maxBal \triangleq Max(\{m[5][1] : m \in 1bMsgs(b, i, Q)\}, \leq) \\
\quad \text{IN } \;\text{CHOOSE } v \in V \cup \{None\} : \exists\, m \in 1bMsgs(b, i, Q) : \\
\qquad\qquad \land\; m[5][1] = maxBal \land m[5][2] = v
\end{array}
$$

The leader of ballot b sends $2a$ messages when it is able to determine a safe value (*i.e.* when it receives $1b$ messages from a quorum), and only if it has not done so before.

$$
\begin{array}{l}
Phase2a(b, i, v) \triangleq \\
\quad \land\; \neg(\exists\, m \in network : m[1] = \text{"2a"} \land m[2] = i \land m[3] = b) \\
\quad \land\; \exists\, Q \in Quorums : \\
\qquad \land\; \forall\, a \in Q : \exists\, m \in 1bMsgs(b, i, Q) : m[2] = a \\
\qquad \land\; \text{LET } maxV \triangleq MaxVote(b, i, Q) \\
\qquad\qquad\quad safe \triangleq \text{IF } maxV \neq None \text{ THEN } \{maxV\} \text{ ELSE } propCmds \\
\qquad\quad\; \text{IN } \;\land\; v \in safe \\
\qquad\qquad\quad \land\; network' = network \cup \{\langle \text{"2a"}, i, b, v \rangle\} \\
\quad \land\; \text{UNCHANGED } \langle propCmds, ballot, vote \rangle
\end{array}
$$

$$
\begin{array}{l}
Vote(a, b, i) \triangleq \\
\quad \land\; ballot[a] = b \\
\quad \land\; \exists\, m \in network : \\
\qquad \land\; m[1] = \text{"2a"} \land m[2] = i \land m[3] = b \\
\qquad \land\; vote' = [vote \text{ EXCEPT } ![a] = [@ \text{ EXCEPT } ![i] = \\
\qquad\qquad [@ \text{ EXCEPT } ![b] = m[4]]]] \\
\quad \land\; \text{UNCHANGED } \langle propCmds, ballot, network \rangle
\end{array}
$$

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\; \exists\, c \in V : Propose(c)
\end{array}
$$

$\lor\ \exists\, b \in \mathit{Ballots} : \mathit{Phase1a}(b)$
$\lor\ \exists\, a \in \mathit{Acceptors},\ b \in \mathit{Ballots},\ v \in V : \mathit{Phase1b}(a,\ b,\ v)$
$\lor\ \exists\, b \in \mathit{Ballots},\ i \in \mathit{Instances},\ v \in V : \mathit{Phase2a}(b,\ i,\ v)$
$\lor\ \exists\, a \in \mathit{Acceptors},\ b \in \mathit{Ballots},\ i \in \mathit{Instances} : \mathit{Vote}(a,\ b,\ i)$

$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{propCmds},\, \mathit{ballot},\, \mathit{vote},\, \mathit{network} \rangle}$

$\mathit{MultiPaxos} \triangleq$ INSTANCE $\mathit{MultiPaxos}$

THEOREM $\mathit{Spec} \Rightarrow \mathit{MultiPaxos}!\mathit{Spec}$

MODULE *DistributedGFPaxos*

A distributed specification of *GFPaxos*, using *DistributedMultiPaxos.tla*.

EXTENDS *MultiConsensus*, *Objects*

VARIABLES
$\quad ballots, votes, network, propCmds$

ASSUME $Instances \subseteq Nat \setminus \{0\}$

ASSUME $Commands = V$

$DistMultiPaxos(o) \triangleq$ INSTANCE *DistributedMultiPaxos* WITH
$\quad ballot \leftarrow ballots[o],$
$\quad vote \leftarrow votes[o],$
$\quad network \leftarrow network[o]$

Is instance i of object o complete?

$Complete(o, i) \triangleq$
$\quad \exists v \in V : DistMultiPaxos(o)!MultiPaxos!Chosen(i, v)$

The next undecided instance for object o:

$NextInstance(o) \triangleq$
$\quad$ LET $completed \triangleq \{i \in Instances : Complete(o, i)\}$
$\quad$ IN IF $completed \neq \{\}$
$\quad\quad$ THEN $Max(completed, \leq) + 1$
$\quad\quad$ ELSE $Max(Instances, \geq)$ the minimum instance

$Msgs \triangleq DistMultiPaxos($CHOOSE $o \in Objects :$ TRUE$)!Msgs$

A type invariant.

$TypeInv \triangleq$
$\quad \land\ ballots \in [Objects \rightarrow [Acceptors \rightarrow \{-1\} \cup Ballots]]$
$\quad \land\ votes \in [Objects \rightarrow [Acceptors \rightarrow$
$\quad\quad\quad [Instances \rightarrow$
$\quad\quad\quad\quad [Ballots \rightarrow \{None\} \cup V]]]]$
$\quad \land\ network \in [Objects \rightarrow$ SUBSET $Msgs]$
$\quad \land\ propCmds \subseteq V$

$InitBallot \triangleq [a \in Acceptors \mapsto -1]$
$InitVote \triangleq [a \in Acceptors \mapsto [i \in Instances \mapsto [b \in Ballots \mapsto None]]]$

The initial state.

$Init \triangleq$
$\quad \land\ ballots = [o \in Objects \mapsto InitBallot]$
$\quad \land\ votes = [o \in Objects \mapsto InitVote]$

$\wedge\ propCmds = \{\}$
$\wedge\ network = [o \in Objects \mapsto \{\}]$

The actions.

$Propose(c) \triangleq$
$\quad \wedge\ propCmds' = propCmds \cup \{c\}$
$\quad \wedge\ \text{UNCHANGED } \langle ballots, votes, network \rangle$

$Phase1a(c) \triangleq$
$\quad \wedge\ \exists\, bs \in [Objects \rightarrow Ballots] :$
$\qquad network' = [o \in Objects \mapsto$
$\qquad\quad \text{IF } o \in AccessedBy(c)$
$\qquad\quad \text{THEN } network[o] \cup \{\langle \text{"1a"}, bs[o] \rangle\}$
$\qquad\quad \text{ELSE } network[o]]$
$\quad \wedge\ \text{UNCHANGED } \langle ballots, votes, propCmds \rangle$

$Phase1b(o, a, c) \triangleq$
$\quad \wedge\ \exists\, b \in Ballots : DistMultiPaxos(o)!Phase1b(a, b, c)$
$\quad \wedge\ \forall\, obj \in Objects \setminus \{o\} : \text{UNCHANGED } \langle ballots[obj], votes[obj], network[obj] \rangle$

The $Phase2a(c)$ action.

$NextInstance$ could be computed from the $1b$ messages. For simplicity, we reuse the $NextInstance(\_)$ operator.

$Phase2a(c) \triangleq$
$\quad \wedge\ \forall\, o \in AccessedBy(c) : \exists\, b \in Ballots :$
$\qquad DistMultiPaxos(o)!Phase2a(b, NextInstance(o), c)$
$\quad \wedge\ \forall\, o \in Objects \setminus AccessedBy(c) : \text{UNCHANGED } \langle network[o] \rangle$
$\quad \wedge\ \text{UNCHANGED } \langle propCmds, ballots, votes \rangle$

$Vote(a, c) \triangleq$
$\quad \wedge\ \forall\, o \in AccessedBy(c) : \exists\, b \in Ballots, i \in Instances :$
$\qquad DistMultiPaxos(o)!Vote(a, b, i)$
$\quad \wedge\ \forall\, o \in Objects \setminus AccessedBy(c) : \text{UNCHANGED } votes[o]$
$\quad \wedge\ \text{UNCHANGED } \langle ballots, network, propCmds \rangle$

$Next \triangleq$
$\quad \exists\, c \in Commands : Propose(c) \vee Phase1a(c) \vee Phase2a(c)$
$\qquad \vee\ \exists\, a \in Acceptors, o \in Objects : Phase1b(o, a, c) \vee Vote(a, c)$

$Spec \triangleq Init \wedge \Box[Next]_{\langle ballots, votes, network, propCmds \rangle}$

$GFPaxos \triangleq \text{INSTANCE } GFPaxos$

$\text{THEOREM } Spec \Rightarrow GFPaxos!Spec$

The spec above cannot be used with TLC because TLC does not accept statements like $fun[x]' = y$ (updating the value of a function on just a subset of its domain), and that's what happens when we reuse the specification of *MultiPaxos*. Below is a second version of the spec, which should be equivalent to the one above, and which can be model-checked with TLC.

$$
\begin{array}{l}
Phase1b2(o,\ a,\ c) \triangleq \\
\quad \land\ \exists\, b \in Ballots : \\
\qquad\quad \land\ ballots[o][a] < b \\
\qquad\quad \land\ \langle \text{“1a”},\ b \rangle \in network[o] \\
\quad \land\ \text{LET } obal \triangleq \\
\qquad\qquad \text{CHOOSE } b \in Ballots : \\
\qquad\qquad\quad \land\ ballots[o][a] < b \\
\qquad\qquad\quad \land\ \langle \text{“1a”},\ b \rangle \in network[o] \\
\qquad \text{IN} \\
\qquad\quad \land\ ballots' = [obj \in Objects \mapsto \\
\qquad\qquad\quad \text{IF } obj = o \\
\qquad\qquad\quad\ \text{THEN } [ballots[o] \text{ EXCEPT } ![a] = obal] \\
\qquad\qquad\quad\ \text{ELSE } ballots[obj]] \\
\qquad\quad \land\ network' = [obj \in Objects \mapsto \\
\qquad\qquad\quad \text{IF } obj = o \\
\qquad\qquad\quad\ \text{THEN } network[o] \cup \\
\qquad\qquad\qquad \{\langle \text{“1b”},\ a,\ i,\ obal,\ DistMultiPaxos(o)!MaxAcceptorVote(a,\ i) \rangle : i \in Instances\} \\
\qquad\qquad\quad\ \text{ELSE } network[obj]] \\
\quad \land\ \text{UNCHANGED } \langle votes,\ propCmds \rangle \\
\quad \land\ \exists\, b \in Ballots : \\
\qquad\quad DistMultiPaxos(o)!Phase1b(a,\ b,\ c)
\end{array}
$$

$$
\begin{array}{l}
Phase2a2(c) \triangleq \\
\quad \text{LET } OkForObj(o,\ b,\ Q) \triangleq \\
\qquad\quad \land\ \neg(\exists\, m \in network[o] : m[1] = \text{“2a”} \land m[2] = NextInstance(o) \land m[3] = b) \\
\qquad\quad \land\ \forall\, a \in Q : \exists\, m \in DistMultiPaxos(o)!1bMsgs(b,\ NextInstance(o),\ Q) : m[2] = a \\
\quad \text{IN} \\
\qquad \land\ propCmds \neq \{\} \\
\qquad \land\ \forall\, o \in AccessedBy(c) : \exists\, b \in Ballots,\ Q \in Quorums : OkForObj(o,\ b,\ Q) \\
\qquad \land\ \text{LET } qs \triangleq [o \in AccessedBy(c) \mapsto \text{CHOOSE } q \in Ballots \times Quorums : \\
\qquad\qquad\qquad\qquad OkForObj(o,\ q[1],\ q[2])] \\
\qquad\qquad\quad safe \triangleq [o \in AccessedBy(c) \mapsto \\
\qquad\qquad\qquad \text{LET } maxV \triangleq DistMultiPaxos(o)!MaxVote(qs[o][1],\ NextInstance(o),\ qs[o][2]) \\
\qquad\qquad\qquad \text{IN } \text{ IF } maxV \neq None \text{ THEN } \{maxV\} \text{ ELSE } propCmds] \\
\qquad\quad \text{IN } \ network' = [o \in Objects \mapsto \\
\qquad\qquad\quad \text{IF } o \in AccessedBy(c) \\
\qquad\qquad\quad\ \text{THEN} \\
\qquad\qquad\qquad \text{IF } c \in safe[o] \\
\qquad\qquad\qquad\ \text{THEN } network[o] \cup \{\langle \text{“2a”},\ NextInstance(o),\ qs[o][1],\ c \rangle\} \\
\qquad\qquad\qquad\ \text{ELSE } network[o] \cup \{\langle \text{“2a”},\ NextInstance(o),\ qs[o][1],\ \text{CHOOSE } v \in safe[o] : \text{TRUE} \rangle\} \\
\qquad\qquad\quad\ \text{ELSE } network[o]]
\end{array}
$$

$\wedge$ UNCHANGED $\langle propCmds, ballots, votes \rangle$
$\wedge \forall o \in AccessedBy(c) : \exists b \in Ballots :$
$\quad DistMultiPaxos(o)!Phase2a(b, NextInstance(o), c)$

$$
\begin{array}{l}
Vote2(a, c) \triangleq \\
\quad \wedge \ \forall o \in AccessedBy(c) : \exists i \in Instances : \\
\qquad \exists m \in network[o] : m[1] = \text{"2a"} \wedge m[2] = i \wedge m[3] = ballots[o][a] \wedge m[4] = c \\
\quad \wedge \ \text{LET } is \triangleq [o \in AccessedBy(c) \mapsto \\
\qquad\qquad \text{CHOOSE } i \in Instances : \\
\qquad\qquad\quad \exists m \in network[o] : m[1] = \text{"2a"} \wedge m[2] = i \wedge m[3] = ballots[o][a] \wedge m[4] = c] \\
\qquad \text{IN} \\
\qquad \wedge \ votes' = [o \in Objects \mapsto \\
\qquad\qquad \text{IF } o \in AccessedBy(c) \\
\qquad\qquad \text{THEN } [votes[o] \text{ EXCEPT } ![a] = [@ \text{ EXCEPT } ![is[o]] = [@ \text{ EXCEPT } ![ballots[o][a]] = c]]] \\
\qquad\qquad \text{ELSE } votes[o]] \\
\quad \wedge \ \text{UNCHANGED } \langle ballots, network, propCmds \rangle
\end{array}
$$

$$
\begin{array}{l}
Next2 \triangleq \\
\quad \exists c \in Commands : Propose(c) \vee Phase1a(c) \vee Phase2a2(c) \\
\qquad \vee \ \exists a \in Acceptors, o \in Objects : Phase1b2(o, a, c) \vee Vote2(a, c)
\end{array}
$$

$$Spec2 \triangleq Init \wedge \Box[Next2]_{\langle ballots, votes, network, propCmds \rangle}$$

Model-checking results:

Configuration: 2 objects, 2 commands (one accessing both objects, one accessing only one object), 3 acceptors, majority quorums, 2 ballots, 2 instances per object.

Verified that $Spec2 \Rightarrow GFPaxos!Spec$. Running on 48 *Xeon* cores with 120*GB* of memory, it took 13 minutes. Result:

> Model checking completed. No error has been found. Estimates of the probability that *TLC* did not check all reachable states because two distinct states had the same fingerprint: calculated (optimistic): $val = 1.8E - 6$ based on the actual fingerprints: $val = 1.3E - 6$
>
> 32992499 states generated, 1026307 distinct states found, 0 states left on queue. The depth of the complete state graph search is 30.