Multicoordinated Paxos

Lásaro Camargos
Unicamp, USI

Rodrigo Schmidt
EPFL, USI

Fernando Pedone
USI

December, 2006

Abstract

Algorithm-oriented: Paxos is a round-based distributed consensus algorithm. In stable runs, proposals are sent to the round leader, which can get a value decided in two communication steps more. Fast Paxos extends Paxos, allowing fast rounds as well as classic ones. In fast rounds, a decision can be learned in two communication steps without relying on a leader but requiring bigger quorums. In this work, we extend Fast Paxos and allow yet another sort of round in which multiple quorums of coordinators can be used. Such new rounds have the same expected latency and quorum size requirements as classic rounds but do not rely on a single leader, providing better availability and allowing load balance. Decentralized rounds like fast ones and ours have a different liveness requirement: the absence of *collisions*—which may happen if different values are proposed concurrently. We show, however, that collisions are inherently more expensive in fast rounds since they involve extra disk writes. We apply our algorithm to solving Generalized Consensus, a generalization of agreement problems such as consensus that allows semantic information to be taken into account when identifying collisions. For that we extend Generalized Paxos, an algorithm for Generalized Consensus based on Fast Paxos.

Application-oriented: One of the most important applications of consensus algorithms is the implementation of state-machine replication for building fault-tolerant distributed systems. Consensus is used to decide on a total order in which commands are applied to each replica, ensuring that all replicas perform exactly the same state transitions. The Paxos consensus protocol delivers such commands to the replicas within three communication steps, but requires that all commands pass through a unique leader. Generalized Paxos is an extension of Paxos motivated by the fact that in many systems commands issued concurrently do not interfere with respect to their execution order. If this assumption holds, commands are learned in two communication steps without relying on a unique process. The emphasis on reducing the number of communication steps in Generalized Paxos has its price, though. Should the system tolerate the failure of any minority of the processes, over 3/4 of them must synchronize for commands to be learned during normal execution. Moreover, when two interfering commands are issued concurrently, a *collision* may happen and extra communication steps and disk writes are necessary. We further generalize Paxos, allowing executions with as many communication steps as the original Paxos (three) but relying on quorums of coordinators instead of on a single one. Since different coordinator quorums can be used, the load can be balanced amongst them. Moreover, the use of coordinators requires fewer processes to synchronize in order to get commands learned: if the failure of any minority of the processes should be tolerated, only a majority of them must exchange messages.

Contents

1 Introduction

In the consensus problem, processes must agree on a single value, given a set of proposals. State-machine replication [6] is probably the most important application of consensus. In this approach, a reliable service is implemented by replicating it in several failure-independent processors, where replicas consistently change their states by applying deterministic commands from an agreed sequence. A consensus instance is used to decide on each command in the sequence.

Paxos is an efficient and fault-tolerant consensus protocol originally intended for state-machine replication [7]. During normal execution, a set of proposer processes (e.g., clients) send their proposed commands to an elected leader. Upon the receipt of a command C, the leader selects the next consensus instance to which no command has been proposed and forwards C, under the selected consensus instance, to a set of acceptor processes. Acceptors "accept" C and send a notification to a set of learner processes (e.g., replicas). Learners learn the decision of a consensus instance when they receive a notification coming from a quorum of acceptors. To tolerate the failure of the leader, each instance of consensus is further subdivided into rounds, explained later. Instances and rounds are similar in definition but completely different in purpose and the reader must be careful not to confuse them.

Even though Paxos provides very good performance, getting commands delivered by replicas in three communication steps, normal execution depends on the availability of the current leader. If the leader fails, its failure must be suspected, a new leader must be elected, and this new leader has to synchronize with a quorum of acceptors before resuming normal execution. These actions take time and may introduce some temporary unavailability to the system. The work in [3] is a good example of how real systems should worry about the failure of the leader in Paxos.

Fast Paxos is an extension of Paxos that admits two modes of execution. The first mode (*classic*) looks exactly the same as the original protocol; the second mode (*fast*) allows proposers to send their proposals directly to the acceptors, reducing the minimum time to get a command learned to two communication steps. Since the leader can be bypassed, its unavailability is much less disruptive to the system. Nonetheless, this advantage comes at a price: acceptor quorums used in the fast execution mode must be significantly bigger than those in the original Paxos algorithm.

Our approach introduces another execution mode to Paxos: *classic multicoordinated*, or simply *multicoordinated*, in which proposers send their pro-

posals to multiple coordinators instead of a single leader. Coordinators behave similarly to the leader in the original Paxos algorithm, but acceptors only accept a proposal if it is forwarded by a quorum of coordinators for the same instance of consensus. After accepting a value, acceptors send a notification to the learners, which learn values correctly notified by a quorum of acceptors. The multicoordinated execution mode has the same latency as the classic one and acceptor quorums have the same size requirements. However, since there are multiple quorums of coordinators, a single process failure does not prevent commands from being learned.

In the fast execution mode, acceptors might accept commands from different proposers for the same instance of consensus. In the worst-case scenario, no quorum of acceptors will accept the same command and learners will not be able to learn anything based on the received notifications. Our approach can have a similar problem if coordinators forward different commands for the same instance of consensus, since acceptors will not be able to accept any value. This problem, called a *collision*, has many possible solutions, but all incur extra communication steps and, in the fast execution mode, extra disk writes.

In real applications, though, not all commands must be applied in the same order to all replicas. This notion of commutable commands can be used to alleviate the problem of collisions since commands can be forwarded or accepted out of order as long as they commute. The Generalized Consensus problem is a generalization of consensus that can take the application semantics into account, like the notion of commutable commands. Moreover, in this problem, learners can augment their learned data structures and, thus, a single instance is enough to implement state-machine replication. Efficient implementations can allow different values (usually sequences) to be learned, as long as they respect the defined semantics.

Generalized Paxos is an extension of Fast Paxos to efficiently solve Generalized Consensus. Our main contribution consists of an extension of this algorithm to allow multicoordinated execution. Since this paper extends previous works on Paxos, Section 2 presents the whole hierarchy of algorithms ours depend upon. Our algorithms for the consensus and generalized consensus problems are presented in Section 3. Practical issues like liveness, collision recovery, and disk writes are discussed in Section 4. Section 5 concludes the paper and compares our approach to others.

2 Paxos: Classic, Fast, and Generalized

2.1 Classic Paxos

The Paxos algorithm originally presented in [7] implements an arbitrary replicated state machine based on consensus. In this section, we focus on the Paxos consensus protocol as explained in [8].

2.1.1 The Problem

The consensus problem can be described in terms of agreement among a set of *learner* processes, on values proposed by a set of *proposer* processes [11]. In the context of a distributed application, *proposers* can be thought of as clients issuing commands and *learners* as the application servers that execute the commands. Clients might also be learners to know whether their issued commands were accepted by the system.

The safety requirements of consensus are three:

Nontriviality: Any value learned must have been proposed.

Stability: A learner can learn at most one value.[1]

Consistency: Two different learners cannot learn different values.

The liveness requirement is less obvious, since it should not prevent progress if any subset of proposers or learners fails. Recall that such roles can be assigned to clients and we cannot require that they not fail. Therefore, another set of processes, the *acceptors*, is necessary to make reliability assumptions about. We call a *quorum* any finite set of acceptors that is large enough to ensure liveness, and define the liveness requirement of consensus as follows:

Liveness: For any proposer p and learner l, if p, l, and a quorum Q of acceptors are nonfaulty and p proposes a value, then l eventually learns some value.

We assume an asynchronous crash-recovery model in which processes communicate by exchanging messages with no bounds for the time it takes for messages to be transmitted or actions to be executed. Messages can be lost or duplicated but not corrupted; processes can fail by stopping only and never perform incorrect actions. Processes are assumed to have some

[1] Stability is usually omitted, since it is tacitly assumed.

sort of local stable storage to keep their state in between failures so that finite periods of absence are not distinguishable from excessive slowness. Although we assume processes may recover, they are not obliged to do so once they have failed. For simplicity, a process is considered to be nonfaulty iff it never fails.

The well-known FLP result [5] implies that under such circumstances no consensus algorithm can ensure the Liveness property if quorums are such that for every acceptor a, there is a quorum Q not containing a—in simpler words, if quorums are defined to tolerate the single failure of any acceptor. As a result, fault-tolerant algorithms must make extra assumptions about the system. We describe later the extra assumptions made by the algorithms we present.

2.1.2 The Solution

The Paxos consensus algorithm executes multiple rounds, sometimes called *ballot numbers*. It assumes an unbounded number of them, totally ordered by a relation $<$. Rounds do not have to be natural numbers or integers, but some optimizations are possible if, for each round i, there is a round $NextRound(i)$ such that no round j satisfies $i < j < NextRound(i)$ (c.f., last paragraphs of Section 2.2). For simplicity, it can be assumed that round numbers correspond to the set of natural numbers, unless we define it differently (Section 4.4).

For each round, an acceptor can "accept" at most one value and the purpose of a round is to get a value accepted by a quorum of acceptors, a situation in which we say the value has been *chosen*. The algorithm guarantees that if a value is chosen at some round, no other round will ever choose a different value. Therefore, a learner can safely learn a value v as soon as it knows that v has been chosen. To prevent two values from being chosen the following assumption is required, stating that quorums of acceptors have non-empty intersections.

Assumption 1 (Quorum Requirement) *If Q and R are acceptor quorums, then $Q \cap R \neq \emptyset$.*

In fact, any general algorithm for asynchronous consensus must satisfy a similar requirement, as shown in [11] (c.f., Accepting Lemma). A simple way to ensure this is defining quorums as any majority of the acceptors.

Although there is a total order among rounds, their execution does not have to follow it, and actions referring to different rounds can even be interleaved. A round is divided into two phases, each one involving two actions.

To orchestrate round executions, Paxos assumes a set of *coordinator* processes, besides proposers, acceptors, and learners. Every round has a single coordinator, responsible for executing the first action of each round phase. The algorithm has also actions referring to the proposal and learning of a value. In the following, we describe each atomic action of the Paxos consensus algorithm:

Propose(p, v) Executed by proposer p when it wants to propose value v. It sends a $\langle$"`propose`", $v\rangle$ message to all round coordinators.

Phase1a(c, i) Executed by the coordinator c of round i. To start round i, c sends a message $\langle$"`1a`", $i\rangle$ to each acceptor a asking a to take part in round i.

Phase1b(a, i) Executed by acceptor a when it receives a message $\langle$"`1a`", $i\rangle$. The action takes place only if i is greater than any other round a has ever heard of, where a has heard of j if actions *Phase1b*(a, j) or *Phase2b*(a, j) have been executed. In this case, a sends a message $\langle$"`1b`", i, *vval*, *vrnd*$\rangle$ to the coordinator of round i, where *vrnd* is the highest-numbered round in which a has accepted a value (or an invalid round number if no value has been accepted by a) and *vval* is the value it accepted in *vrnd*. The pre-condition of this action makes sure that after it is executed for round i, acceptor a will not execute it for a round j such that $j < i$. As we show in action *Phase2b*, this action also prevents a from accepting a value for a round j lower than i. This is a guarantee to the coordinator of i that the pair *vval* and *vrnd* will remain consistent as the information about the latest value accepted by a for a round number lower than i.

Phase2a(c, i) Executed by the coordinator c of round i after it receives a "`1b`" message for round i coming from each acceptor in a quorum. c sends a message $\langle$"`2a`", i, *val*$\rangle$ to the acceptors, where *val* is c's selected proposal defined as follows. If no "`1b`" message informed of a previously accepted value, then c is free to select *val* among the proposals received directly from proposers. Otherwise, c must pick up a value that has been or might be chosen in a previous round to make sure that no two different values will end up being chosen. This procedure actually gives Paxos an interesting property: If a value v is chosen at round j, then no acceptor will ever accept a value other than v at a round greater than j. Therefore, c must consider only the "`1b`" messages with the highest value of *vrnd*. Moreover, since Paxos ensures

that no two acceptors can accept different values at the same round, all such messages are guaranteed to have the same value *vval* and c picks it up. Hereinafter we use the term *pick up a value* when referring to the proposal selection performed by coordinator c during a *Phase2a* action.

Phase2b(a, i) Executed by acceptor a when it receives a $\langle$"`2a`"$, i, val\rangle$ message. If a has not heard of a round j greater than i then it accepts *val* and sends a message $\langle$"`2b`"$, i, val\rangle$ to all learners. As we can notice, the fact that an acceptor only accepts values at round i sent by the coordinator of i in a phase "`2a`" message (which is the same sent to all acceptors) ensures that no two acceptors accept different values at the same round.

Learn(l) Executed by learner l when it receives a message $\langle$"`2b`"$, i, val\rangle$ from each acceptor in a quorum. The messages imply that *val* has been chosen and l can learn it.

If different coordinators keep starting new rounds, it may happen that no value is ever chosen since no action *Phase2b* action will be executed. To ensure liveness, a distinguished coordinator must be selected as a leader, a position that entitles it to start new rounds. When there is a single leader in the system, it will be able to start a round that is high enough to overcome all previously started rounds and make it succeed. However, this is just a liveness condition; safety is never violated no matter how many coordinators incorrectly think to be the leader.

Since a coordinator sends the value to be accepted only in the beginning of phase 2, the first phase of the algorithm can be executed before receiving any proposal. On a real application, probably many consensus instances will be needed and the leader can execute phase 1 "a priori" for all consensus instances. Thus, the amortized latency for solving each instance becomes only three messages steps if there are no failures and no other coordinator interferes by starting a higher-numbered round.

2.2 Fast Paxos

It takes at least three communication steps for a proposal to be learned in the original Paxos algorithm (hereinafter called Classic Paxos): one step for the proposal to reach the leader and two more for the second phase of the leader's current round. Fast Paxos [10] is an extension of the classic algorithm in

which proposers can send their proposals directly to the acceptors, reducing the latency of reaching a decision in one communication step.

The first difference between Classic Paxos and Fast Paxos is that, in the latter, each round has its own set of quorums and we call a quorum for round i an i-quorum. The reason for this is made clear later in this section (see Assumption 2). As a second difference, Fast Paxos has two sorts of round: *classic* and *fast*. Classic rounds have the same structure as rounds in Classic Paxos. Fast rounds share the same first phase as classic rounds, but their second differs slightly.

In a fast round i, after receiving the "`1b`" messages from an i-quorum, if the coordinator is free to pick any value, it can send a special value *Any* to the acceptors. After receiving the "`2a`" message with the value *Any*, acceptor a waits for a message $\langle$"`propose`"$, v\rangle$, in case it has not received one yet, and behaves as if it had received a message $\langle$"`2a`"$, i, v\rangle$ from the coordinator of round i. Note that for the principle of Fast Paxos to work, proposers should send their "`propose`" messages to both coordinators and acceptors. Acceptors are still bound to accept just one value per round but, differently from Classic Paxos, different acceptors can accept different values in the same fast round. This means that the rule used by the coordinator of round i to pick up a value after receiving the "`1b`" messages from an i-quorum must be revisited.

The heart of the Paxos algorithm (classic or fast) lies in ensuring that if a value v is chosen at round i, no acceptor will have accepted a value different from v at any round j such that $j > i$. This property is guaranteed by the first phase of each round and the rule used by the coordinator to pick up the value to be sent in the "`2a`" message. We say that a value v is *pickable* at round i iff no other value was or can still be chosen at any round j such that $j < i$.

As explained in the previous section, picking up a value in Classic Paxos is relatively simple: it suffices to look at the *vval* field of any "`1b`" message with the greatest value for *vrnd*. To understand the situation with Fast Paxos, let us assume the coordinator of round i has just received the "`1b`" messages for i from an i-quorum Q. If no message has a valid round in the *vrnd* field, no value has been or might be chosen at lower-numbered rounds and any proposed value is pickable. Otherwise, let k be the greatest value for *vrnd* received amongst the phase "`1b`" messages. If all messages in which $vrnd = k$ report the same value v as *vval*, it might be the case that v was or will be chosen at a round $j < k$ (recall the property stated in the previous paragraph). Moreover, since any k-quorum must intersect Q and acceptors a in Q have guaranteedly executed action $Phase1b(a, i)$ for round i ($i > k$),

no value different from v can be chosen at k. Therefore, the coordinator can safely pick v up.

Now consider the case in which more than one value have been reported in the phase "`1b`" messages with $vrnd = k$ (k still being the greatest value for $vrnd$ reported in the "`1b`" messages). This implies that no value was chosen or will be chosen at a round lower than k. As a result, the coordinator must only figure out which of the values has been or might yet be chosen at k. There are three cases to consider with respect to the "`1b`" messages received with $vrnd = k$:

1. There is no value v and k-quorum R such that, for every acceptor a in $Q \cap R$, a message $\langle$"`1b`"$, k, v\rangle$ was received from a. This implies that no value v has been or might be chosen at k since no k-quorum has even partially agreed on a value v at k. In this case, any proposed value is pickable.

2. There is only one value v such that, for some k-quorum R, a message $\langle$"`1b`"$, k, v\rangle$ has been received from every acceptor in $Q \cap R$. This means that only value v has been or might be chosen at k depending on what the other acceptors have accepted or might still accept. In this case, v is pickable.

3. There are two different values v and w and two k-quorums R and S such that, for every acceptor a in $Q \cap R$, a message $\langle$"`1b`"$, k, v\rangle$ was received from a, and, for every acceptor b in $Q \cap S$, a message $\langle$"`1b`"$, k, w\rangle$ was received from b. This means that either one of the values has been chosen or might still be chosen depending on what the other acceptors in R and S accept at k. By the quorum requirement, R and S have a non-empty intersection, which prevents both values from being chosen, but this intersection does not intersect Q and i's coordinator cannot decide on which value is pickable.

The way to avoid the third case to happen is by strengthening the assumption made on the intersection of quorums and making sure that the intersection of any two quorums R and S as shown in case 3 above also intersects Q. If this is ensured, the situation discussed in case 3 will never happen and the coordinator can stick to the solutions for cases 1 and 2 only. Therefore, Fast Paxos relies on the following quorum requirement:

Assumption 2 (Fast Quorum Requirement) *For any rounds i and j:*

- *If Q is an i-quorum and S is a j-quorum, then $Q \cap S \neq \emptyset$.*

- *If Q is an i-quorum, R and S are j-quorums, and j is fast, then $Q \cap R \cap S \neq \emptyset$.*

In the general case, this stronger assumption requires bigger quorums. If every set of $n - E$ acceptors is a quorum for a fast round (fast quorum, for short) and every set of $n-F$ acceptors is a quorum for a classic round (classic quorum), where n is the total number of acceptors, then n must be greater than $2E + F$ as well as greater than $2F$. These constraints are achieved, for example, if every set of $\lceil (2n + 1)/3 \rceil$ acceptors is a fast and classic quorum. If classic quorums are defined to be any majority of acceptors, fast quorums must be as big as $\lceil (3n + 1)/4 \rceil$ acceptors. It has been shown, however, that any asynchronous consensus protocol that allows a decision to be reached in two communication steps must satisfy similar quorum requirements [11] (Fast Learning Theorem).

If two different values v_1 and v_2 are proposed at the same fast round i, it may be that none gets chosen, even in the absence of failures or suspicions, because of *collisions* [10]. A collision happens when the acceptors of a fast quorum accept different values. In pessimistic scenarios, collisions will prevent any value from being chosen in a fast round. This happens, for example, if half of the acceptors accepts v_1 and the other half accepts v_2. There are three ways to break the tie and recover from a collision, and all reduce to executing a higher-numbered round. However, depending on how this new round is chosen, latency can be reduced considerably.

Let us assume a collision has happened at round i. The simplest approach has i's coordinator c to monitor the acceptors' phase "`2b`" messages and start a new round from the beginning after it learns the collision has happened. This approach is expensive as it takes four communication steps to recover. However, if c is also the coordinator of round $i + 1$, then c can exploit the fact that the "`2b`" messages sent for round i can be interpreted as "`1b`" messages for round $i + 1$, and proceed directly to the second phase of round $i+1$, incurring only two communication steps for collision recovery. This second approach is called *coordinated recovery*.

As an extension of coordinated recovery, if round i's "`2b`" messages are also sent to the set of acceptors, they can try to guess the coordinator's "`2a`" message for round $i + 1$. They do that by interpreting these "`2b`" messages for round i as "`1b`" messages for round $i + 1$ and applying the same algorithm as the coordinator does to pick up a value for a phase "`2a`" message. As seen before, this algorithm is guaranteed to return a pickable value. However, there is no guarantee that the acceptors will pick up the

same value and this requires that round $i + 1$ be fast, allowing acceptors to accept different values. As explained in [10], some strategies can be used to try to make them accept the same value. The advantage of this method is that it takes only one communication step to recover from collisions. This third approach is called *uncoordinated recovery*.

2.3 Generalized Paxos

Collisions are the main problem with Fast Paxos because they may happen even in stable periods of execution (i.e., no failures or suspicions). In a state machine replication scenario, a collision may happen if two commands are proposed concurrently to the same instance of consensus. In many systems, however, commands may commute and there is no need for totally ordering them since the final state is the same independently of the order in which they are applied. Consensus, as applied to state machine replication, is too strong to capture this notion and a collision may happen even if the two concurrent proposals are commutable.

Generalized Consensus is a generalization of the consensus problem, defined in terms of a data structure called *command structure*, or simply *c-struct*. Depending on the set of c-structs defined, generalized consensus represents a different problem. As explained in [9], one can define c-struct sets for traditional consensus, total order broadcast, generic broadcast [13], etc. C-structs are general enough to capture semantic information like the notion of commutable commands. An efficient implementation of Generalized Consensus can use this to mitigate the problem of collisions. Generalized Paxos does exactly that: It extends Fast Paxos to solve Generalized Consensus and provides very good performance for certain c-struct sets.

2.3.1 C-Struct Sets

We use basically the same definitions and notation as the work in [9]. A c-struct set *CStruct* is defined in terms of an element $\bot$, a set of commands *Cmd*, an operator $\bullet$ that appends a command to a c-struct, and a set of axioms listed later. They are very general data structures. For example, one could create a c-struct set where c-structs are subsets of *Cmd*, $\bot$ is the empty set, and $v \bullet C$ simply adds element C to the current value of v. Another c-struct set could have c-structs as partially ordered sets, $\bot$ as the empty set, and $v \bullet C$ as an operation that extends partially ordered set v with command C by making C succeed (with respect to the partial order) any conflicting element of v, given an external conflicting relation over *Cmd*—a

c-struct set that could capture the notion of commutable commands.

Before we present the five axioms of a c-struct set, some definitions are necessary. A finite sequence with elements C_i is represented by $\langle C_1, C_2, \ldots, C_m \rangle$. $Seq(S)$ is defined to be the set of all (finite) sequences whose elements are in the set S (with possible repetitions of elements in the sequence). Moreover, we use the term c-seq when referring to a finite sequence of commands—that is, an element of $Seq(Cmd)$. We can now extend the operator $\bullet$ to sequences of commands as follows:

$$v \bullet \langle C_1, \ldots, C_m \rangle = \begin{cases} v & \text{if } m = 0, \\ (v \bullet C_1) \bullet \langle C_2, \ldots, C_m \rangle & \text{otherwise} \end{cases}$$

We define that c-struct w extends c-struct v ($v \sqsubseteq w$) iff there exists a c-seq σ such that $w = v \bullet \sigma$. Given a set T of c-structs, we say that c-struct v is a lower bound of T iff $v \sqsubseteq w$ for all w in T. A greatest lower bound (glb) of T is a lower bound v of T such that $w \sqsubseteq v$ for every lower bound w of T, and we represent it by $\sqcap T$. Similarly, we say that v is an upper bound of T iff $w \sqsubseteq v$ for all w in T. A least upper bound (lub) of T is an upper bound v of T such that $v \sqsubseteq w$ for every upper bound w of T, and we represent it by $\sqcup T$. If $\sqsubseteq$ is a reflexive partial order on the set of c-structs and a glb or lub of T exists, then it is unique. For simplicity of notation, we use $v \sqcap w$ and $v \sqcup w$ to represent $\sqcap\{v, w\}$ and $\sqcup\{v, w\}$, respectively. Two c-structs v and w are defined to be *compatible* iff they have a common upper bound, and a set S of c-structs is compatible iff its elements are pairwise compatible.

We say that c-struct v is *constructible from* a set P of commands if $v = \bot \bullet \sigma$, for some c-seq σ containing all elements of P. Moreover, we say that v *contains* command C if v is constructible from some set P of commands such that $C \in P$. We define $Str(P)$ to be the set of all c-structs constructible from subsets of P for some set P of commands—that is, $Str(P) \triangleq \{\bot \bullet \sigma : \sigma \in Seq(P)\}$.

A c-struct set *CStruct* must satisfy axioms CS0-CS4 below. CS0-CS2 are basic requirements to satisfy the properties discussed above. CS3 and CS4 are necessary for Generalized Paxos and similar algorithms to ensure the safety and liveness properties of Generalized Consensus, described next.

CS0. $\forall C \in Cmd, w \in CStruct : w \bullet C \in CStruct$

CS1. $CStruct = Str(Cmd)$

CS2. $\sqsubseteq$ is a reflexive partial order on $CStruct$.

CS3. For any set $P \subseteq Cmd$ and any c-structs u,v, and w in $Str(P)$:

- $v \sqcap w$ exists and is in $Str(P)$.
- If v and w are compatible, then $v \sqcup w$ exists and is in $Str(P)$.
- If $\{u, v, w\}$ is compatible, then u and $v \sqcup w$ are compatible.

CS4. For any command $C \in Cmd$ and compatible c-structs v and w in $CStruct$, if v and w both contain C then $v \sqcap w$ contains C.

2.3.2 Generalized Consensus

We can now generalize the original definition of consensus to deal with a c-struct set instead of single absolute values. The problem is defined in terms of a c-struct set $CStruct$ which, as shown in the previous section, is based on a null value $\bot$, a set Cmd of commands, and an operator $\bullet$. Proposers propose commands in Cmd and we let $learned[l]$ be learner l's currently learned c-struct (initially $\bot$). Generalized Consensus is defined by the following properties:

Nontriviality: For any learner l, $learned[l]$ is always a c-struct constructible from some of the proposed commands.

Stability: For any learner l, if the value of $learned[l]$ at any time is v, then $v \sqsubseteq learned[l]$ at all later times.

Consistency: The set $\{learned[l] : l \text{ is a learner}\}$ is always compatible.

Liveness: For any proposer p and learner l, if p, l, and a quorum Q of acceptors are nonfaulty and p proposes a command C, then $learned[l]$ eventually contains C.

Different instances of the problem are created by different $CStruct$ sets. In [9], Lamport presents c-struct sets that define traditional consensus, total order broadcast, generic broadcast, among others.

2.3.3 The Generalized Paxos Algorithm

Generalized Paxos is an extension of Fast Paxos to solve Generalized Consensus. The algorithm has the advantage that, by the problem definition, a collision is not characterized if two acceptors accept different but compatible c-structs. In such case, both acceptors can later extend their accepted c-structs so that they converge to the same one (since compatible c-structs

have a common upper bound). C-struct sets like command histories with commutable commands, explained better in Section 3.3, might have very few incompatible c-structs, which reduces the chances of a collision to happen and favors the use of fast rounds.

Generalized Paxos relies on the Fast Quorum Requirement (Assumption 2). It assumes that acceptors have previously accepted $\bot$ at a round 0 lower than any other round, and has the following atomic actions:

Propose(p, C) The same as in Fast Paxos, except that C must be a command in *Cmd*.

Phase1a(c, i), *Phase1b*(a, i) The same as in Classic Paxos, except that values are now c-structs.

Phase2Start(c, i) When coordinator c of round i receives a "`1b`" message for round i coming from each acceptor in an i-quorum Q, it has to pick up a value to send back to the acceptors. To understand the action, we must revisit some definitions and properties of Paxos. We define a c-struct v to be chosen (and possibly learned) iff there is a round j at which every acceptor in some j-quorum has accepted an extension of v. The algorithm keeps the property that if v has been chosen at some round j and w is accepted by some acceptor at a higher-numbered round, then $v \sqsubseteq w$. As before, for coordinator c to gather the set of all possibly chosen c-structs in previous rounds, it suffices to look at the "`1b`" messages with the highest-numbered *vrnd* value. Let k be such a round number. There are only two cases to consider.

First, if there is no k-quorum R such that, for every acceptor a in $R \cap Q$, c has received a "`1b`" message from a with $vrnd = k$, then c is assured that no c-struct different from $\bot$ has been or might be chosen at k. Moreover, the algorithm ensures that, if a c-struct v has been chosen at a round $j < k$, then any value w accepted at k satisfies $v \sqsubseteq w$. Therefore, c can pick any c-struct received in one of the "`1b`" messages in which $vrnd = k$.

If the first case does not apply, then, for every k-quorum R such that c has received a "`1b`" message with $vrnd = k$ from every acceptor in $R \cap Q$, c calculates the glb of the c-structs *vval* received in such messages and adds it to a set Γ initially empty. After that, Γ will contain all c-structs that have been or might be chosen at lower-numbered rounds. The second condition of the Fast Quorum Requirement ensures that

Γ is compatible and, therefore, has a least upper bound $\sqcup\Gamma$ that can be safely picked by c.

After picking up a c-struct *val* based on the previous two cases, c sends a message $\langle$"`2a`", i, *val*$\rangle$ to all acceptors.

Phase2aClassic(c, i) This action appends commands to a c-struct previously proposed by the coordinator c of round i. It is executed by coordinator c only if it has already sent a phase "`2a`" message for round i to the acceptors. It could be executed for a fast round i, although it makes more sense to be executed only if i is classic. Let $\langle$"`2a`", i, *val*$\rangle$ be the latest phase "`2a`" message c has sent for round i and let *newval* be $val \bullet \sigma$ for some c-seq σ of proposed values received in "`propose`" messages from proposers. c simply sends a message $\langle$"`2a`", i, *newval*$\rangle$ to all acceptors.

Phase2bClassic(a, i) Executed by acceptor a when it receives a message $\langle$"`2a`", i, *val*$\rangle$. Let k be the highest-numbered round a has heard of and v be the latest value accepted by a in k ($\bot$ if none); if $i > k$ or $i = k$ and $v \sqsubset val$, a accepts *val* and sends message $\langle$"`2b`", i, *val*$\rangle$ to every learner.

Phase2bFast(a, i) This action is enabled iff i is fast, i is the highest-numbered round a has heard of, a has already accepted a value in i, and a has received a $\langle$"`propose`", $C\rangle$ message. Let v be the latest value a has accepted in i; a accepts $v \bullet C$ and sends message $\langle$"`2b`", $i, v \bullet C\rangle$ to every learner.

Learn(l) Executed by learner l after it receives a phase "`2b`" message for some round i from each acceptor in an i-quorum. Let v be the glb of the values received in such messages; l sets $learned[l]$ to $learned[l] \sqcup v$.

In Generalized Paxos, any round starts by the round coordinator executing *Phase1a*(c, i). Acceptors then should execute action *Phase1b*(a, i), followed by the execution of *Phase2Start*(c, i) by c and *Phase2aClassic*(a, i) by the acceptors. After this point, the execution will depend on whether i is fast or classic. If i is classic, proposers propose, the coordinator continuosly executes *Phase2aClassic*(c, i), followed by acceptors executing *Phase2bClassic*(a, i) and, then, learners executing *Learn*(l). If i is fast, proposers propose, acceptors execute *Phase2bFast*(a, i), and learners execute *Learn*(l).

3 Multicoordinated Paxos

Fast rounds do not depend on a single coordinator during normal execution, but do have stricter requirements on acceptor quorum sizes. In our approach, we extend classic rounds to have multiple coordinators, making them more reliable while maintaining their latency and acceptor quorum requirements.

In this section, we introduce these multi-coordinated rounds by first extending the Fast Paxos consensus protocol of Section 2.2. We then present our our Generalized Consensus protocol. Finally, we present our protocol applied to solve the Generic Broadcast problem, so that the reader can understand how it works in a more general instance of Generalized Consensus.

We provide the correctness proof and a formal TLA$^+$ specification of our generalized consensus protocol in the appendix of this document. The other protocols' correctness and formal specifications can be easily derived from them.

3.1 A Consensus Implementation

As in the original Fast Paxos, in the extended protocol rounds are still divided into classic and fast, but we relax the assumption that each round has a single coordinator—note that, in this sense, a classic round differs from original rounds in Classic Paxos. We define a quorum of coordinators for a round i, or an i-coordquorum for short, as any set of coordinators satisfying Assumption 3, below. We say that c is a coordinator of round i if it belongs to any i-coordquorum.

Assumption 3 (Coord-quorum requirement) *For any two quorums of coordinators P and Q for the same classic round, $P \cap Q \neq \emptyset$.*

Any coordinator can execute phases "`1a`" and "`2a`", but acceptors will accept a value only if sent by all coordinators in an i-coordquorum. It is easy to see that original Classic Paxos rounds (with a single coordinator) are classic rounds with a single one-element quorum of coordinators. Fast rounds can have multiple coordinators and Assumption 3 would place no restriction upon them. However, since fast rounds are meant to avoid coordinators during normal execution, we see no reason to have something different from a single coordinator for them except in some very specific scenarios, but we defer this discussion to a latter section on collision recovery. The algorithm has the following actions:

Propose(p, v) The same as in Fast Paxos.

Phase1a(c, i), *Phase2a*(c, i) The same as in Classic/Fast Paxos (since the Fast Paxos rule to pick up a value is used in action *Phase2a*), except that c can be any coordinator of i.

Phase1b(a, i) The same as in Classic Paxos, except that a sends the "`1b`" message to all coordinators of round i.

Phase2b(a, i) Executed by acceptor a, for round i. This action is enabled if a as has not heard of a round greater than i and has received a message $\langle$"`2a`", i, *val*$\rangle$ coming from all coordinators in some i-coordquorum with the same value *val*. If i is a fast round and $val = Any$, then a can accept any value sent in a "`propose`" message. If i is classic and, therefore, $val \neq Any$, then a accepts *val*. After accepting value v, a sends the message $\langle$"`2b`", $i, v\rangle$ to all learners.

Learn(l) The same as in Classic Paxos.

The main difference between this protocol and Fast Paxos is that, as described, *Phase2b* action handles "`2a`" messages from different coordinators for the same round, ensuring that it only accepts a value that has been proposed by a whole coordinator quorum. Because coordinators only forward single values to be accepted, the handling of these values is actually quite simple. It gets more complicated when coordinators and acceptors must cope with incremental values, as in generic broadcast protocols. This is shown in the next section. We postpone the discussion about liveness of Multicoordinated Paxos to Section 4.3.

3.2 The Generalized Algorithm

We now explain the complete Multicoordinated Generalized Paxos algorithm. Rounds are defined as in the previous section and the algorithm assumes a c-struct set *CStruct*, the Fast Quorum Requirement (Assumption 2) for quorums of acceptors, and the Coord-quorum Requirement (Assumption 3) for quorums of coordinators.

As in Generalized Paxos, we ensure that if a c-struct v is chosen at a round i, then any c-struct w that is accepted by any acceptor at some round $j > i$ extends v ($v \sqsubseteq w$). This is guaranteed by the first phase of a round due to the rule used by the coordinators to pick up a value based on the phase "`1a`" messages received by a quorum of acceptors (explained in Section 2.3.3, action *Phase2Start*(c, i)). In this section, we embody the rule in function *ProvedSafe*$(Q, 1bMsg)$ defined below, where Q is a quorum

of acceptors and *1bMsg* is a mapping from every acceptor a in Q to a phase "`1b`" message. In previous sections, we did not need to name the fields of a "`1b`" message and just considered the structure $\langle$"`1b`", i, *vval*, *vrnd*$\rangle$. In order to explain function *ProvedSafe*, though, we have to name its fields.We assign the following names to the four fields of a phase "`1b`" message, in order: *type*, *rnd*, *vval*, and *vrnd*.

Definition 1 *For any set of acceptors Q, and mapping 1bMsg from each acceptor in Q to a phase "`1b`" message, let:*

- $vals(S) \triangleq \{1bMsg[a].vval : a \in S\}$
  Set of vval values sent by acceptors in $S \subseteq Q$.
- $vrnds \triangleq \{1bMsg[a].vrnd : a \in Q\}$
  Set of vrnd values sent in all "`1b`" messages.
- $k \triangleq Max(vrnds)$
  Highest-numbered round in vrnds.
- $kacceptors \triangleq \{a \in Q : 1bMsg[a].vrnd = k\}$
  Set of acceptors that sent "`1b`" messages with vrnd equal to k.
- $QinterR \triangleq \{Q \cap R : R \text{ is a } k\text{-quorum } \}$
  Set of intersections between Q and every k-quorum R.
- $QinterRAtk \triangleq \{S \in QinterR : S \subseteq kacceptors\}$
  Intersections of interest: those in which all elements sent "`1b`" messages with vrnd equal to k.
- $\Gamma \triangleq \{\sqcap(vals(inter)) : inter \in QinterRAtk\}$
  glb's of the values sent in the "`1b`" messages, for every intersection of interest.

Then ProvedSafe is defined as follows:

$$ProvedSafe(Q, 1bMsg) \triangleq \text{IF } QinterRAtk = \{\} \text{ THEN } vals(kacceptors) \text{ ELSE } \{\sqcup\Gamma\}$$

ProvedSafe$(Q, 1bMsg)$ returns a set of c-structs that are pickable for round i if Q is an i-quorum and every acceptor a in Q has sent "`1b`" message $1bMsg[a]$ with field *rnd* equal to i. It follows exactly the same rule as explained in Section 2.3.3 (c.f., action *Phase2Start*). In simpler words, if there is no k-quorum R for which all acceptors in $R \cap Q$ have sent "`1b`"

messages for round i with field *vrnd* equal to k, then any value that has been reported in "`1b`" messages with $vrnd = k$ are pickable. Otherwise, the Fast Quorum Requirement ensures that set Γ is compatible. As a result, its lub exists and is pickable.

To make our algorithm description precise, we must explain the variables required by each process. Proposers need no internal variables. Coordinators keep only their current round and the latest c-struct they have sent to the acceptors at that round in a phase "`2a`" message. The variables of a coordinator c are the following:

crnd$[c]$ The current round of c. Initially 0.

cval$[c]$ The latest c-struct c has sent in a phase "`2a`" message for round *crnd*$[c]$. Initially $\bot$.

An acceptor a keeps three variables:

rnd$[a]$ The current round of a, that is, the highest-numbered round a has heard of. Initially 0.

vrnd$[a]$ The round at which a has accepted the latest value. Initially 0.

vval$[a]$ The c-struct a has accepted at *vrnd*$[a]$. Initially $\bot$.

Each learner l keeps only the c-struct it has learned so far.

learned$[l]$ The c-struct currently learned by l. Initially $\bot$.

In the following we present the basic atomic actions that compose the algorithm. They always ensure safety, but some restrictions must be imposed on their execution for the algorithm to ensure liveness as well. We postpone the discussion about practical issues such as ensuring liveness or dealing with collisions to Section 4.

Propose(p, C) Executed by proposer p when it wants to propose command C. It sends a $\langle$"`propose`", $C\rangle$ message to acceptors and round coordinators.

Phase1a(c, i) Executed by coordinator c, for round i. It is enabled iff

- c belongs to an i-coordquorum and
- $crnd[c] < i$.

It sends message $\langle$"`1a`", $i\rangle$ to the acceptors.

Phase1b(a, i) Executed by acceptor a, for round i. It is enabled iff

- $rnd[a] < i$ and
- a has received a message $\langle$"`1a`", $i\rangle$.

The action sets $rnd[a]$ to i and sends message $\langle$"`1b`", i, $vval[a]$, $vrnd[a]\rangle$ to the coordinators of round i.

Phase2Start(c, i) Executed by coordinator c, for round i. It is enabled iff

- $crnd[c] < i$ and
- c has received a phase "`1b`" message for round i from every acceptor in an i-quorum Q.

First, it picks some value v in *ProvedSafe*$(Q,$ *1bMsg*$)$, where *1bMsg* is a mapping from every acceptor a in Q to the phase "`1b`" message c received from a. Then, it sets $cval[c]$ to v, $crnd[c]$ to i, and sends message $\langle$"`2a`", $i, v\rangle$ to the acceptors.

Phase2aClassic(c) Executed by coordinator c. It is enabled iff c has received a message $\langle$"`propose`", $C\rangle$. It sets $cval[c]$ to $cval[c] \bullet C$ and sends message $\langle$"`2a`", $crnd[c]$, $cval[c]\rangle$ with the updated value of $cval[c]$ to the acceptors.

Phase2bClassic(a, i) Executed by acceptor a, for round i. It is enabled iff

- $rnd[a] \leq i$,
- a has received a phase "`2a`" message for round i from every coordinator c in an i-coordquorum L, and
- $vrnd[a] < i$ or $vval[a]$ is compatible with $\sqcap$*L2aVals*, where *L2aVals* is the set of all c-structs received in the "`2a`" messages for round i from the coordinators in L.

If $vrnd[a]$ equals i, it sets $vval[a]$ to $vval[a] \sqcup (\sqcap$*L2aVals*$)$; otherwise, it sets $vval[a]$ to $\sqcap$*L2aVals*. Then, it sets $vrnd[a]$ and $rnd[a]$ to i (if this is not the case yet), and sends message $\langle$"`2b`", i, $vval[a]\rangle$ with the updated value of $vval[a]$ to the learners.

Phase2bFast(a) Executed by acceptor a. It is enabled iff

- $rnd[a]$ is a fast round,
- $rnd[a] = vrnd[a]$, and

- a has received a message $\langle$"`propose`", $C\rangle$.

It sets $vval[a]$ to $vval[a] \bullet C$ and sends message $\langle$"`2b`", $vrnd[a], vval[a]\rangle$ with the updated value of $vval[a]$ to the learners.

Learn(l) Executed by learner l. It is enabled iff a has received phase "`2b`" messages for some round i from an i-quorum Q. It sets $learned[l]$ to $learned[l] \sqcup (\sqcap Q2bVals)$, where $Q2bVals$ is the set of all c-structs received in the "`2b`" messages for round i from acceptors in Q.

In a general execution scenario, one (or more) coordinators will execute action *Phase1a*(c, i) for some high enough round i. Acceptors will acknowledge it by executing action *Phase1b*(a, i). All coordinators in i-coordquorums will then execute *Phase2Start*(c, i), which will trigger the execution of *Phase2bClassic*(a, i) by the acceptors. This sequence of actions happens only when a new round starts, which is supposed to happen seldom, due to failures or collisions. During the rest of the round, a simpler execution pattern takes place. If the round is fast, proposers execute *Propose*(p, C), acceptors execute *Phase2bFast*(a), and learners execute *Learn*(l), multiple times. If the round is classic (multicoordinated or not), proposers execute *Propose*(p, C), the round coordinators execute *Phase2aClassic*(c), acceptors execute *Phase2bClassic*(a, i), and learners execute *Learn*(l), also repeatedly.

In [9], Lamport discusses how to deal efficiently with large c-structs and all the ideas presented there can be directly applied to our algorithm. The complexity of calculating lubs, glbs, and verifying the compatibility of c-structs will depend on the c-struct set being used. Most c-struct sets we are aware of (e.g., those presented in [9]) admit relatively simple implementations of these operations. The complexity of calculating function *ProvedSafe* also depends on how quorums are defined and it can be simplified if quorums are defined as any set of processes of a certain size (e.g., majority sets).

3.3 A Generic Broadcast Implementation

In the Generic Broadcast problem [13], processes must agree on a partially ordered set, or poset, of proposed commands. The partial order must order non-commutable commands, where commutable is defined in terms of a conflict relation $\asymp$. In other other words, if two commands C and D are non-commutable ($C \asymp D$) and both belong to the poset, then $C \prec D$ or $D \prec C$. Read only operations are common examples of commutable commands. Operations changing the same piece of data, as a file in a file

system or a row in a database, may be commutable or not, depending on the application.

We refer to the posets in the generic broadcast problem as command histories and define the $\bullet$ operator to append a command to a command history according to the conflict relation $\asymp$. $\bullet$ is defined for sequences of commands as the ordered application of $\bullet$ to each command in the sequence. More formally, if Cmd is the set of commands that can be broadcast by processes, $(S, \prec)$ is a command history defined by the set of commands $S \subseteq Cmd$ and the partial order $\prec$, and $\langle C_1, \ldots, C_m \rangle$ is a sequence of commands in Cmd, then:

$$(S, \prec) \bullet C = \begin{cases} (S, \prec) & \text{if } C \in S, \\ (S \cup \{C\}, \prec_\bullet) : & \forall a, b \in S, a \prec b \Leftrightarrow a \prec_\bullet b \\ & \forall a \in S, a \asymp C \Rightarrow a \prec_\bullet C \quad \text{otherwise} \end{cases}$$

and

$$(S, \prec) \bullet \langle C_1, \ldots, C_m \rangle = \begin{cases} (S, \prec) & \text{if } m = 0, \\ ((S, \prec) \bullet C_1) \bullet \langle C_2, \ldots, C_m \rangle & \text{otherwise} \end{cases}$$

We say that command history g extends command history h ($h \sqsubseteq g$) if there exists a sequence σ of commands such that $g = h \bullet \sigma$. As a matter of fact, command histories are c-structs, and therefore $\sqcap$, $\sqcup$, and "compatible", defined on Section 3.2, are also defined for command histories. Moreover, we use $\bot$ to represent an empty command history.

By letting $learned[l]$ be the command history that learner l has learned at some point in time, we formally define the properties of generic broadcast as follows.

Non-triviality For any learner l, $learned[l]$ only contains proposed commands.

Stability For any learner l, the value of $learned[l]$ at any time is an extension of $learned[l]$ at any previous time.

Consistency The set $\{learned[l] : l \text{ is a learner}\}$ is always compatible.

Liveness For any proposer p and learner l, if p, l, and a quorum Q of acceptors are nonfaulty, and p proposes a command C, then $learned[l]$ eventually contains C.

We now extend the multicoordinated protocol from the previous section to solve the generic broadcast problem. In this new algorithm, as new values

are proposed, acceptors will add new values to their previously accepted command histories, and increasing prefixes of these command histories are learned as they become chosen.

To simplify the algorithm's presentation, we define quorums in terms of the cardinalities of the sets. For that, let n be the number of acceptors in the system, F be the maximum number of acceptor failures that does not prevent progress, and E be the maximum number of acceptor failures that still allows fast termination. Acceptor quorums are defined as any set of at least $n - F$ acceptors, and fast acceptor quorums are defined as any set of at least $n - E$ acceptors. As explained in Section 2.2, as long as $2E + F < n$, Assumption 1 and Assumption 2 are satisfied. As for coordinators, we let any set with a majority of coordinators be a quorum, what trivially satisfies Assumption 3.

Also for the sake of simplicity, we formalize the variables informally introduced in the consensus algorithm. Because the kind of values handled by these algorithms are different (single values in the first and command histories in the second), some variables have different types and initial values; their semantics, however, remains the same.

The coordinator keeps two variables:

$crnd[c]$ The current round of coordinator c. Initially 0.

$cval[c]$ The latest command history c has sent in a phase "`2a`" message for round $crnd[c]$ or *none*, if no value has been sent by c to the acceptors in its current round. It is initially $\bot$ for all coordinators.

An acceptor a keeps three variables:

$rnd[a]$ The current round of a, that is, the highest-numbered round a has heard of. Initially 0.

$vrnd[a]$ The round at which a has accepted a value for the last time. Initially 0.

$vval[a]$ The command history a has accepted at round $vrnd[a]$. Initially $\bot$.

Accordingly to the acceptors' initialization, learners will always learn command histories with $\bot$ as the smallest element. Therefore, $learned[l]$ initially equals $\bot$, for every learner in the system.

The following actions define the generic broadcast protocol. Therein we consider generic implementations of command histories or, more generally, c-structs. We give an efficient implementation of command histories and the $\bullet$ operator to this protocol at the Section 3.3.2.

Propose(p, C) Executed by proposer p to propose a new command C. It sends the message $\langle$"`propose`", $C\rangle$ to acceptors and coordinators.

Phase1a(c, i) Executed by any coordinator c to start round i. This action is enabled iff:

- c is in some i-coordquorum and
- $crnd[c] < i$.

The action sends a message $\langle$"`1a`", $i\rangle$ to all acceptors asking them to take part in round i.

Phase1b(a, i) Executed by acceptor a, for round i. The action is enables iff:

- $rnd[a] < i$.
- a has received a message $\langle$"`1a`", $i\rangle$

It sets $rnd[a]$ to i and sends a message $\langle$"`1b`", $i, vval[a], vrnd[a]\rangle$ to all coordinators of round i. The pre-condition of this action makes sure that after it is executed for round i, acceptor a will not execute it for a round j such that $j \leq i$.

Phase2Start(c, i) Executed by any coordinator c at round i. This action is enabled iff:

- $crnd[c] < i$ and
- c received a "`1b`" message for round i from all acceptors in a set Q of $n - F$ acceptors.

It sends a message $\langle$"`2a`", $i, val\rangle$ to the acceptors, where *val* is the value that c selected by looking in the "`1b`" messages coming from acceptors in Q. *val* must be an extension of any command history that may have been decided in a round $j < i$. The procedure to select *val* is given at Section 3.3.2, after we have given an implementation of a command history, in Section 3.3.1.

Phase2Start sets $cval[c]$ to *val* and $crnd[c]$ to i and, due to the action's pre-condition, is executed only once per *round*.

Phase2aClassic(c) Executed by coordinator c to extend its value for round $crnd[c]$. It is enabled iff c has received a $\langle$"`propose`", $C\rangle$ message. This action sets $cval[c]$ to $cval[c] \bullet C$ and sends a $\langle$"`2a`", $crnd[c], cval[c]\rangle$ message with the updated $cval[c]$.

Phase2bClassic(a, i) Executed by acceptor a in round i. This action is enabled iff:

- $rnd[a] \leq i$,
- if i is a fast round, then a received a "`2a`" message for round i from some coordinator c; if i is a classic round, then a received a "`2a`" message for round i from all coordinators in some set L with a majority of the coordinators, and
- $vrnd[a] < i$ or $vval[a]$ is compatible with $\sqcap L2aVals$, where $L2aVals$ is the set of values received in the "`2a`" messages from coordinators in L.

If $vrnd[a]$ equals i, it sets $vval[a]$ to $vval[a] \sqcup (\sqcap L2aVals)$; otherwise, it sets $vval[a]$ to $\sqcap L2aVals$. Then, it sets $vrnd[a]$ and $rnd[a]$ to i (if this is not the case yet), and sends message $\langle$"`2b`", $i, vval[a]\rangle$ with the updated value of $vval[a]$ to the learners.

Phase2bFast(a) Executed by acceptor a, and enabled iff:

- $rnd[a]$ is a fast round,
- $rnd[a] = vrnd[a]$, and
- a has received a $\langle$"`propose`", $C\rangle$ message.

The action sets $vval[a]$ to $vval[a] \bullet C$ and sends a $\langle$"`2b`", $vrnd[a], vval[a]\rangle$ message to all learners with the variables' updated values.

Learn(l) Executed by learner l. It is enabled iff:

- l has received "`2b`" messages for some round i from all acceptors in some set Q of acceptors,
- if i is a classic round, then Q has cardinality $n - F$,
- if i is a fast round, then Q has cardinality $n - E$, and

It sets $learned[l]$ to $\sqcup(learned[l] \sqcap Q2bVals)$, where $Q2bVals$ is the set of values received in the "`2b`" messages received from acceptors in Q.

3.3.1 A Simple Command History Representation

Command history can be represented as sequences of commands. The command history $\bot$, for example, may be represented simply as $\langle\rangle$, while the

command history $\bot \begin{array}{l} \nwarrow a \nwarrow c \\ \nwarrow b \nwarrow d \end{array}$, where the arrows point to the previous elements in the partial order, may be represented as $\langle a, b, c, d\rangle$, $\langle a, c, b, d\rangle$, $\langle a, b, d, c\rangle$, $\langle b, d, a, c\rangle$, $\langle b, a, d, c\rangle$, or $\langle b, a, c, d\rangle$. In the last example, because the sequences do not represent all the ordering information, the conflict relation ($\asymp$) is still needed to assess the order of commands in the sequence.

New commands can be added to a sequence by simply appending it at the sequence's end, if it is not in the sequence yet. Formally, the $\bullet$ operator can be defined as follows:

$$\langle c_1, \ldots, c_m\rangle \bullet C = \begin{cases} \langle c_1, \ldots, c_m\rangle & \text{if } \exists i, C = c_i \\ \langle c_1, \ldots, c_m, C\rangle & \text{otherwise.} \end{cases}$$

To determine the longest common prefix between two two command histories H and I, the following operator checks if the head of H exists on I before any conflicting command, in which case it is part of their common prefix. Otherwise, the operator recursively proceeds on the tail of H stripped of the descendants of its head.

$$\begin{array}{l} \mathit{Prefix}(H, I) \triangleq \\ \quad \text{IF } H = \langle\rangle \vee I = \langle\rangle \\ \quad \text{THEN } \langle\rangle \\ \quad \text{ELSE IF } \exists j : \wedge\ \mathit{Head}(H) = I[j] \\ \qquad\qquad\qquad\ \wedge \neg\exists k < j : \mathit{Head}(H) \asymp I[k] \\ \qquad\quad \text{THEN } \langle \mathit{Head}(H)\rangle \circ \mathit{Prefix}(\mathit{Tail}(H), I \setminus \mathit{Head}(H)) \\ \qquad\quad \text{ELSE } \mathit{Prefix}(\mathit{Tail}(H) \setminus \mathit{Descendants}(\mathit{Head}(H), \mathit{Tail}(H)), I) \end{array}$$

Observe that in this definition we extrapolated the use of the set minus operator, $\setminus$, to remove some element from a sequence. Although this definition resembles TLA$^+$, it is not a correct definition in such language because, besides the unconventional use of $\setminus$, TLA$^+$ does not allow the definition of recursive operators, and some rewriting is required for this definition to conform with that language's syntax. Nonetheless, we keep this definition for the sake of simplicity.

To calculate the glb of a set of command histories instead of a simple pair, the search on the sequence I could be performed in parallel in many sequences. However, we stick to an iterative approach of simpler understanding.

$$\begin{array}{l} \sqcap S \triangleq \text{IF } S = \{e\} \\ \qquad\quad \text{THEN } e \\ \qquad\quad \text{ELSE LET } e, f \in S, e \neq f \end{array}$$

$$\text{IN} \quad \sqcap (\mathit{Prefix}(e, f) \cup S \setminus \{e, f\})$$

Determining if two sequences are compatible is more complicated. The procedure presented below iterates over the first sequence, H, looking for the first element e of H that does not appear in I. If, during this search, some conflicting ordering is identified among the sequences—in a procedure similar to the one on the Prefix operator—, the *AreCompatible* operator identifies that the sequences are not compatible. If no incompatibility is found, then the procedure searches for descendants of e in I. If some exists, then it also indicates incompatibility, as e would have to appear before its descendant also in I. If none is found, the operator recursively proceeds on the rest of H, but keeping the list of removed elements in a set of ancestors A, so that new descendants can be identified in the next steps.

$$\begin{array}{l}
\mathit{AreCompatible}(H, I, A) \triangleq \\
\quad \text{IF } H = \langle\rangle \vee I = \langle\rangle \\
\quad\ \text{THEN TRUE} \\
\quad\ \text{ELSE IF } \exists j : \wedge\ \mathit{Head}(H) \asymp I[j] \\
\qquad\qquad\qquad\ \wedge \neg\exists k < j : \mathit{Head}(H) = I[k] \\
\qquad\ \text{THEN FALSE} \\
\qquad\ \text{ELSE IF } \exists j : \mathit{Head}(H) = I[j] \\
\qquad\qquad\ \text{THEN IF } \exists f \in A : \mathit{Head}(H) \asymp f \\
\qquad\qquad\qquad\ \text{THEN FALSE} \\
\qquad\qquad\qquad\ \text{ELSE } \mathit{AreCompatible}(\mathit{Tail}(H), I \setminus \mathit{Head}(H), A) \\
\qquad\qquad\ \text{ELSE } \mathit{AreCompatible}(\mathit{Tail}(H), I, A \cup \{\mathit{Head}(H)\})
\end{array}$$

AreCompatible can be rewritten to generate the lub of two sequences as it goes on verifying their compatibility. This operator would be more complex, though, and we opted for presenting its simplified version, which assumes that the sequences are compatible.

$$\begin{array}{l}
H \sqcup I \triangleq \text{IF } H = \langle\rangle \\
\qquad\qquad\ \text{THEN } I \\
\qquad\qquad\ \text{ELSE IF } \exists j : \mathit{Head}(H) = I[j] \\
\qquad\qquad\qquad\ \text{THEN } \langle \mathit{Head}(H) \rangle \circ (\mathit{Tail}(A) \sqcup B \setminus \mathit{Head}(A)) \\
\qquad\qquad\qquad\ \text{ELSE } \langle \mathit{Head}(H) \rangle \circ (\mathit{Tail}(A) \sqcup B)
\end{array}$$

Finally, the following operator calculates the lub of a set of compatible sequences.

$$\begin{array}{l}
\sqcup S \triangleq \text{IF } S = e \\
\qquad\qquad\ \text{THEN } e
\end{array}$$

ELSE LET $e, f \in S : e \neq f$
IN $\sqcup((e \sqcup f) \cup (S \setminus \{e, f\}))$

3.3.2 Selecting *val* in *Phase2a*

Once an implementation of command histories has been picked we can define how a coordinator c picks the value to be forwarded to acceptors on action *Phase2Start* in some round i. Thes procedure is described below for command histories implemented by sequences. In the description, we refer to the third and fourth fields of a message $m = \langle$"`1b`"$, i, vval[a], vrnd[a]\rangle$ sent by acceptor a for round i as $m.vval$ and $m.vrnd$. Let

- Q be a set of acceptors of cardinality $n - F$, such that c has received a "`1b`" message from each acceptor in Q, for round i.
- $1bQ$ be the set of "`1b`" messages received by c, from all acceptors in Q, in round i.
- Let k be the biggest $m.vrnd$ among all messages $m \in 1bQ$.
- Let k-*acceptors* be the set of acceptors in Q from which c has received a message m such that $m.vrnd = k$.
- Let $vals(S)$ be the set $m.vval$ for all messages $m \in 1bQ$ received from an acceptor in a set S.

If k is a classic round, then any subset of k-*acceptors* with $n-2F$ elements could combine with the acceptors from which messages were not received to form quorum R. In this case, the acceptors in R could have chosen any prefix of the values accepted by the acceptors in $R \cap Q$. Let *InterAtk* be the set of all such subsets, that is, subsets of k-*acceptors* with cardinality $n - 2F$. If *InterAtk* is empty, then c can choose any message m from an acceptor in k-*acceptors*, and forward any extension of $m.vval$ to the acceptors. Otherwise, c must forward an extension of $\sqcup\Gamma$, where Γ is the longest prefix shared by acceptors in the sets of *InterAtk*, i.e., $\Gamma = \{\sqcap vals(e) : e \in InterAtK\}$. In the case of simple majority quorums, that is, $F = \lfloor (n-1)/2 \rfloor$, $n - 2F = 1$ and Γ will equal the set of values on messages received from all acceptors in k-*acceptors*, and the calculus of glbs can be skipped.

If k is a fast round, then the subsets of k-*acceptors* in *InterAtk* must have cardinality $n - 2E$, as this is the minimum size of an intersection of two fast quorums. The rest of the procedure to pick a sequence to extend remains the same.

4 Practical Issues

4.1 Use of multiple coordinators

As we have mentioned before, Classic Paxos can be seen as an implementation of our algorithm where all rounds are classic and each round i has only one i-coordquorum with a single element (the coordinator of round i in Classic Paxos). We call such rounds single-coordinated as opposed to multicoordinated ones, which are also classic but have multiple quorums of coordinators.

The main problem of single-coordinated rounds as compared to multicoordinated ones has to do with availability. If the coordinator of a single-coordinated round crashes, time must be spent with the indentification of the failure (usually done through timeouts), the election of a new coordinator, and the execution of the first phase of a higher-numbered round, before normal execution can be retaken. The optimization of these tasks may also effect performance or availability. For example, aggressive failure detection may trigger false suspicions, and simple leader election algorithms can elect a crashed process or more than a single leader at a time.

If a round has multiple quorums of coordinators, a single failure will not require immediate round change, avoiding all the aforementioned availability problems. A simple implementation of Multicoordinated Paxos would have a fixed number of coordinator processes in every round and define coordinator quorums of multicoordinated rounds as any majority of them so that the Coord-quorum Requirement is satisfied. In such case, the failure of any minority of the coordinators leaves at least one quorum of coordinators still available and, therefore, able to forward proposals to the acceptors.

One could argue that fast rounds also do not rely on a single coordinator during normal execution, since acceptors can accept proposals directly from proposers. However, the Fast Quorum Requirement imposes stricter restrictions on how fast quorums are defined, which also affects availability since fewer failures are tolerated.

The existence of multiple quorums of both coordinators and acceptors also enables implementations with better load balance than classic paxos. Recall that, in classic paxos, all commands must go through the current leader (round coordinator) and, depending on the system load, this might be a performance bottleneck. In Multicoordinated Paxos, for a command C to be learned in multicoordinated rounds, it must be forwarded by a coordinator quorum and accepted by an acceptor quorum only. If there are multiple coordinator and acceptor quorums, no acceptor or coordinator

needs to process all commands proposed. A simple way to distribute the load has the proposer p of a command C choose (randomly or through some uniformly distributed function) a quorum of coordinators and a quorum of acceptors for C. p sends the "`propose`" message only to the chosen coordinator quorum, with the chosen quorum of acceptors piggybacked in the message since all coordinators in the quorum must forward C to the right acceptors. The coordinators send their "`2a`" message with C only to the indicated quorum of acceptors, which accept C and send phase "`2b`" messages to the learners. If not all learners should learn about C, the same approach can be used, forwarding with C the set of learners to which the phase "`2b`" messages should be sent.

Fast rounds also allow distributing the load over the set of acceptors. As before, however, the stricter quorum requirement implies a worse distribution. If fast rounds are composed of $\lceil (3n+1)/4 \rceil$ acceptors, where n is the total number of acceptors (a necessary condition if any majority of the acceptors is a quorum for a classic round), then it is not hard to verify that every acceptors will have to process more than 3/4 of the proposed commands. In multi-coordinated rounds, if any majority of the coordinators of a round i is an i-coordquorum and any majority of acceptors is an i-quorum, then the load can be distributed so that each coordinator processes at most $(1/2 + 1/nc)$ of the proposed commands, where nc is the total number of coordinators for round i, and each acceptor accepts at most $(1/2 + 1/n)$ of the proposed commands. It is true that in this scenario, each command must be dealt twice (first by the coordinators and then by the acceptors), but the coordinators' action is much cheaper since it must not involve disk writes, as we show later.

Doing this sort of load balancing does not jeopardize availability. The optimistic use of a single quorum only does not mean that the other quorums cannot be used. A clever implementation would resort initially to a single quorum of coordinators and acceptors. If the command is not learned after some time has elapsed (triggered by a timeout or a failure suspicion), then other quorums might be used. This wait time can be set to a minimum since they will never trigger a round change as discussed in the beginning of this section.

Lastly, it is clear from the algorithm that the sets of coordinators and acceptors need not have the same number of elements. Actually, in many cases it might be better to have more acceptors than coordinators in a round. Note that the set of coordinators for round i can be completely different from the set of coordinators of round $j \neq i$, but this is not the case for acceptors since they must be queried in every new round to check whether a value

has already been chosen at some previous round. Moreover, the acceptors' task of accepting a value is more expensive than the coordinators' task of forwarding it, since the former requires a disk write but the latter does not. As a result, implementations might use a high number of acceptors to improve the system's resilience or performance (due to load balancing). But an equally high number of coordinators for a round increases only the availability of that round, and the load balancing will not be as effective since the coordinators' task is cheaper. For a small system, a configuration with 5 acceptors in total and 3 coordinators for multi-coordinated rounds (with different sets of coordinators for different rounds) sounds plausible, since it tolerates the failure of any two processes and does not introduce temporary unavailability if a single coordinator crashes.

4.2 Collisions

Multicoordinated rounds have a drawback that inexists in single-coordinated ones—collisions. In multicoordinated rounds, a collision happens when commands proposed concurrently arrive at the coordinators in different orders and this leads to their forwarding of incompatible c-structs. If no coordinator quorum forwards c-structs whose glb can extend the values previously accepted by the acceptors, the round is stuck since no new command can get accepted.

This is a different type of collision as compared to the one that may occur in fast rounds, explained in Section 2.2. In fast rounds, a collision happens when acceptors accept incompatible c-strucs that cannot further extend the values learned by learners so far. In this case, however, acceptors pay the price of accepting commands that will never be learned, which does not happen in collisions of multicoordinated rounds. This is a major difference between the two kinds of collisions since acceptors must write on stable storage every time they accept a value but coordinators do not have to, as we explain in Section 4.4.

The mechanism to solve collisions in the original Fast Paxos algorithm presented in Section 2.2 cannot be directly applied to Generalized Paxos. The only way to adapt it to Generalized Paxos we are aware of cannot tolerate the failure of any acceptor. If no other algorithm exists, the techniques we present below for multicoordinated rounds can be used at the cost of one extra communication step for the acceptors to identify the collision in the c-structs they have accepted. Another possibility consists of explicitly starting a new higher-numbered classic single-coordinated round from the beginning after its coordinator identifies the collision.

In multicoordinated rounds, collision identification can be done by the acceptors when they receive the phase "2a" messages from the coordinators of a classic round i. If two coordinators of the same i-coordquorum send "2a" messages for round i with incompatible c-structs, acceptors execute action *Phase1b*$(a, i + 1)$ as if they had received a phase "1a" message for round $i+1$. What comes next will depend on whether round $i+1$ is classic or fast.

If round $i+1$ is classic and enough acceptors identify the collision, which will normally happen if messages are not lost and processes do not crash, then the coordinators of round $i+1$ will execute action *Phase2Start*$(c, i + 1)$ based on the received messages, followed by one or more executions of action *Phase2aClassic*(c). Thus, the collision in round i will be resolved with only two extra communication steps (as compared to the usual three of a classic round). Clearly, to avoid that another collision happens when the coordinators start round $i + 1$, it is advisable to have it as a single-coordinated round. After some time of normal execution, if conflicting commands stop being proposed, the coordinator of round $i+1$ can start a multicoordinated round again. This approach is a variation of the *coordinated recovery* presented in [10].

If round $i + 1$ is fast, performance can be improved by setting $i + 1$-coordquorums wisely. Since the Coord-quorum Requirement does not place any restriction when it comes to fast rounds, we can define that any single acceptor by itself constitutes a coordinator quorum for a fast round (playing both roles—acceptor and coordinator). When a coordinator of round $i + 1$ (which is also an acceptor) executes actions *Phase2Start*$(c, i + 1)$ and *Phase2aClassic*(c), it can locally accept the values supposedly sent in the "2a" messages, without actually sending them. This approach can resolve collisions with only one extra communication step. However, new collisions might happen when the round $i + 1$ is started. This is a variation of the *uncoordinated recovery* presented in [10], which also presents some interesting ideas to avoid having collisions when round $i + 1$ is started. We do not cover them here because the use of a fast round to recover from a collision in a classic one does not seem to be of practical use. We just present the idea of the uncoordinated recovery mechanism for completeness.

One might ask herself if there can be some kind of round that does not rely on a single coordinator but also avoids collisions. In the most general case, its existence would contradict the FLP result since quorums could be defined to tolerate a single failure and the absence of collisions would mean that liveness can be achieved in such rounds.

4.3 Liveness

The possibility of starting new rounds allows the algorithm to progress if a round does not succeed (because of coordinator crashes or collisions). However, starting new rounds carelessly is also a problem because new rounds can be continuously initiated without ensuring liveness. In Classic Paxos and Fast Paxos, this is prevented by using some (unreliable) leader election algorithm that eventually elects a single leader that will be responsible for starting a higher-numbered round under its coordination. In Multicoordinated Paxos, we use the same strategy to prevent the continuous initialization of of new rounds. Message losses can also prevent liveness. The solution to that is to have processes keep on re-sending their last message, which can be optimized as described in Section 2.4.1 of [10].

If the current leader starts a new classic single-coordinated round (of which it is the only coordinator), liveness is ensured as long as the leader does not crash and other coordinators do not wrongly think they are the current leader and try to start a higher-numbered round. If other coordinators interfere, the leader must be notified. This is done by extending the algorithm a little, making acceptors reply to phase "`1a`" or "`2a`" messages with a round number lower than their current one in order to notify the coordinator which sent that message that its current round number is too low to get values accepted. When a coordinator that thinks to be the leader receives such a message from the acceptors, it can start a higher-numbered round.

If the leader starts a fast round, liveness is ensured as long as the leader does not crash during the execution of phase 1 and collisions do not happen during the rest of the round execution. If there are no failures, collisions can be resolved by adapting the collision recovery mechanisms presented in [10] to Generalized Paxos, by having acceptors identify collisions and use the approach we explained in Section 4.2 to solve them, or by simply having the leader identify the collision and start a new classic single-coordinated round to solve it.

In classic multicoordinated rounds, liveness is ensured if the leader does not crash during the execution of phase 1 and neither collisions happen nor all quorums of coordinators become unavailable during the rest of the round execution. The failure of the leader is not a problem since another correct leader is eventually selected which will make sure that a new round gets started. As for collisions, the mechanism presented in Section 4.2 can be used—the only restriction we make is that the leader must be one of the coordinators for the following round, otherwise the leader might think

of the round change as an interference and try an even higher-numbered round. Last, to cope with the failure of coordinators, the leader must start a new round if it believes that other coordinators have failed. Their possible failure can be assessed by monitoring their "2a" messages or some external failure detection mechanism. When the leader notices that there are not enough coordinators in the current round to ensure progress, it starts a new higher-numbered round with a different set of coordinator quorums.

4.4 Reducing disk writes

Assumption 3 imposes no restriction between coordinator quorums of different rounds. If it is always possible to start new rounds with any set of coordinator quorums, coordinators are not required to write on stable storage. A coordinator that crashes and later recovers could just be seen as a new coordinator in the system, which is easily implemented by having an "incarnation" counter associated with its identifier. In the following we explain how new rounds can be created with any set of coordinator quorums.

Consider round numbers as a records of the form $\langle Count, Id, RType, S\rangle$, where $Count$ is a natural number, Id is a coordinator's unique identifier, $RType$ is a natural number, and S is a set of coordinator quorums. Counter $Count$ always allows the creation of a new high-numbered round, Id identifies the coordinator that created the round, $RType$ tells the round type (fast if 0, classic otherwise), and S identifies all valid coordinator quorums for this round. A round is uniquely identified by the first three fields, and the total order relation is given by comparing them lexicographically. Field S is merely informative and is not taken into consideration when comparing two rounds. Using this approach, when the current leader wants to start a new round, it can simply define the four fields according to its current knowledge.

Since Assumption 2 requires that quorums of different rounds intersect, acceptors cannot lose their state after a crash and assume a different identity upon recovery. This happens because the values accepted by acceptors cannot be forgotten, or the algorithm's safety would be compromised. Therefore, these values must always be stored in stable storage, incurring a disk write (or equivalent operation) whenever an acceptor executes a *Phase2b* action. As a result, acceptors are not as easily replaceable as coordinators and more complex strategies must be used [12].

Action *Phase1b* also changes the internal state of an acceptor and, at a first sight, this seems to imply that *Phase1b* must also write on disk. However, an acceptor a may store $rnd[a]$ only in main memory as long

as, after recovering from a crash, it manages to initialize $rnd[a]$ with a higher value than the previous one. This is done as follows: Field *Count* described previously in this section can be composed of a major and a minor component, *MCount* and *mCount*. When an acceptor executes *Phase1b* for some round, if *MCount* equals the previous value in $rnd[a]$, it changes $rnd[a]$ in volatile memory only; otherwise, it writes it on disk. During recovery, the acceptor simulates the reception of a "`1a`" message with an *MCount* higher than the one it has on disk. To get values accepted by the recovered acceptor, coordinators will be forced to use higher rounds. In the normal case, acceptors write on disk only once, when they are started. In the presence of failures, this strategy results in one extra disk write at each acceptor, per recovery.

4.5 Setting rounds and quorums

The schema used to define round numbers shown in the previous section, that is, as a vector of the form $\langle MCount{:}mCount, Id, RType, S\rangle$ should fit most of application scenarios. However, there are some specific scenarios for which this schema should be adapted for better performance. There are two main points on doing these adaptations: the likeliness of collisions and how they are recovered, and what type of round follow each other.

For example, if collisions are a constant in the system, then fast rounds should not be used because the recovery cost could sum up bigger than the economy provided by fast rounds. If fast rounds are used in conjunction with coordinated recovery, then they should be followed by single-coordinated rounds. In the case uncoordinated recovery is considered, fast rounds should be followed by multi-coordinated fast rounds. (See Section 4.2.)

In the case of the example, where some fast rounds should be followed by other fast rounds, using the record described above would force the use of rounds with different *Id* or *Count* field. Because recovery relies on a process knowing exactly what is the next round number, this schema would not work. A possible solution is to change how the field *RType* is interpreted (e.g., letting all *RType* in the range 0 to 5 be interpreted as fast, instead of simply 0).

The set of coord-quorums can be defined at run-time, considering the status of the system when a new round is created. To ensure liveness, multi-coordinated rounds should be followed by single-coordinated rounds (See Section 4.3). However, this transition need not be brusque, but could be done through a series of multi-coordinated rounds with smaller quorums, minimizing the risk of collisions while still allowing for the benefits of multi-

coordination.

Below we present a couple of general scenarios and discuss how round numbers and quorums could be defined to them. These scenarios are not necessarily disjoint, and real world systems would probably share the characteristics of both of them, as well as other relevant ones, and a per-case analysis is needed to find the best solution. Notice that none of the Paxos algorithms require the ordered use of all ballot numbers and that coordinators are allowed to skip rounds—possibly based on the the dynamics of the environment it is inserted into. However, because collision recovery explores the sequential execution of rounds, skipping rounds could prevent efficient recovery and therefore the rounds' configuration should be defined *a priori* as precisely as possible.

Clustered systems Clustered systems connected through high-speed networks have a large probability of spontaneously ordering the messages sent to the same destinations. In such systems, acceptors executing a fast round are likely to accept the values in the same order, and values are likely to be learned in two communication steps, even when conflicting proposals are made.

In such a scenario, ranges of infinite fast rounds followed by single-coordinated rounds seems the best configuration: most of the time values are fast learned and, in the rare case of conflicts, they are solved by the variant uncoordinated recovery presented in Section 4.2. Coordinators can always resort to the next single-coordinated round to ensure liveness if uncoordinated recovery does not succeed.

For this scenario, the values of the *RType* field in the basic approach can be mapped to a range of many fast rounds followed by classic rounds, as we mentioned before. By dividing *RType* in a major and minor component, as we did with *Count*, the number of fast rounds can even be infinite.

If conflicts are rare but tend to be persistent and require coordinated recovery, then a solution is to map *RType*'s even values to fast rounds and odd values to single-coordinated classic rounds.

Conflict prone In widely distributed systems or under high load, messages tend to get inverted and non-commutable proposals often result in conflicts. In such environments, if non-commutable commands prevail, then the algorithms will always end up resorting to single-coordinated classic rounds to finish.

In such an environment, it does not make sense to have fast nor multi-

coordinated rounds, and round numbers can be defined as the set of integer and partitioned among a finite set of coordinators by some module function. For an infinite set of coordinators or for having each coordinator as the leader of infinitely many consecutive rounds, a simplified round number of the form $\langle MCount{:}mCount, Id\rangle$ could be used.

5 Conclusion

Multicoordinated Paxos is an extension of Paxos that allows multiple coordinators to be used concurrently to improve availability. Our algorithm also provides means to better balance the load and adapt to changes at runtime. The possibility of collisions makes our approach more attractive if the application semantics can be taken into account. As a result, we have applied it to Generalized Consensus.

Generic Broadcast is an implementation of Generalized Consensus with commutable commands. In the same way the algorithms in [13] are similar to Generalized Paxos, the algorithms in [2] are similar to ours. However, both [13] and [2] consider the crash-stop model only, do not allow changing the round type, and, different from Generalized and Multicoordinated Paxos, may identify a collision even if conflicting commands are received in the same order.

Spontaneous order as a means to solve consensus was first used in [14]. In fact, B-Consensus is a crash-stop simplified version of the algorithm in Section 3.1; it was later adapted to the crash-recovery model in [4], but requiring two disk writes per proposal. Moreover, none of these protocols has a generalized version, and they rely solely on the spontaneous order to ensure liveness, while Multicoordinated Paxos can always switch to a single-coordinated round to ensure progress in the absence of failures.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proc. of the 14th Intl. Conference on Distributed Computing*, pages 268–282, Toledo, Spain, 2000.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th Symp. on Operating System Design and Implementation, OSDI'06*, November 2006.

[4] L. Camargos, E. R. M. Madeira, and F. Pedone. Optimal and practical wab-based consensus algorithms. In *Proc. 12th Intl. Euro-Par Conference, Euro-Par'06*, pages 549–558, Dresden, Germany, 2006.

[5] M. J. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[6] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[7] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

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

[9] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2004.

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

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

[12] M. Massa and L. Lamport. Cheap paxos. In *Proc. of the 2004 Intl. Conference on Dependable Systems and Networks*, June 2004.

[13] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[14] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. of the 4th European Dependable Computing Conference*, pages 44–61, 2002.

A Proof of Correctness

A.1 Preliminaries

Our proofs depend on a number of basic definitions and propositions, presented in this section. Apart from an extra proposition, everything in this section comes from [9], since we want to follow the same notation and proof structure as presented in that paper.

Concerning round numbers (hereinafter called *ballot numbers*), we only assume that they are totally ordered by a relation $<$ and there is a smallest ballot number 0. We use *balnum* as an abbreviation for *ballot number*, and let *BalNum* be the set of all balnums. Balnums can be either fast or classic, but not both. Quorums of acceptors and coordinators depend upon the ballot number. We assume both the Fast Quorum Requirement (Assumption 2) and the Coord-quorum Requirement (Assumption 3). Lastly, we assume a c-struct set *CStruct* upon which Generalized Consensus is defined.

We start by defining a data structure called *ballot array*, used by our abstract algorithms. Ballot arrays keep the votes of each acceptor at each balnum and the ballot number at which each acceptor currently is (the highest-numbered balnum it has heard of). Vote entries are initialized with a value *none* that is not in *CStruct*. Due to that, we extend $\sqsubseteq$ to cope with *none* such that $none \sqsubseteq none$ but $\neg(v \sqsubseteq w)$ if either v or w (but not both) equals *none*.

Definition 2 (Ballot Array—Definition 1 of [9]) *A* ballot array bA *is a mapping that assigns to each acceptor a a balnum $\widehat{bA}_a$ and to each acceptor a and balnum m a value $bA_a[m]$ that is a c-struct or equals none, such that for every acceptor a:*

- $bA_a[0] \neq none$
- *The set of balnums m with $bA_a[m] \neq none$ is finite.*
- $bA_a[m] = none$ *for all balnums* $m > \widehat{bA}_a$.

We say that a value v is *chosen at* balnum m if an m-quorum accepts v at m. We can also define chosen at with respect to a ballot array as follows.

Definition 3 (Chosen at—Definition 2 of [9]) *A c-struct v is* chosen at *balnum m* in *ballot array bA iff there exists an m-quorum Q such that $v \sqsubseteq bA_a[m]$ for all acceptors a in Q. A c-struct v is* chosen in *ballot array bA iff it is chosen at m in bA for some balnum m.*

Considering that an acceptor a can only accept c-structs at balnums equal to or greater than its current one ($\widehat{bA}_a$ in ballot array bA), we define a c-struct to be *choosable at* balnum m iff it is or can still be chosen at m.

Definition 4 (Choosable at—Definition 3 of [9]) *A c-struct v is* choosable at *balnum m* in *ballot array bA iff there exists an m-quorum Q such that $v \sqsubseteq bA_a[m]$ for every acceptor a in Q with $\widehat{bA}_a > m$.*

A c-struct is *safe at* balnum m iff it extends any c-struct choosable at a balnum j such that $j < m$, and we define a ballot array bA to be *safe* iff every entry $bA_a[m]$ different from *none* is safe at m, for every acceptor a. If acceptors accept only safe c-structs at balnums greater than or equal to their current ones, the algorithm guarantees that if v is ever chosen at a balnum i, then no acceptor will have accepted a c-struct w that does not extend v at a balnum j greater than i.

Definition 5 (Safe at—Definition 4 of [9]) *A c-struct v is* safe at m in bA *iff $w \sqsubseteq v$ for every balnum $k < m$ and every c-struct w that is choosable at k. A ballot array bA is* safe *iff for every acceptor a and balnum k, if $bA_a[k]$ is a c-struct then it is safe at k in bA.*

The following proposition states that the chosen values of a safe ballot array are compatible. It implies that an algorithm can satisfy the consistency property of Generalized Consensus by having acceptors accept only safe values at balnums that are no lower than their current ones. Its detailed proof appears in [9].

Proposition 1 (Proposition 1 of [9]) *If a ballot array bA is safe, then the set of values that are chosen in bA is compatible.*

A.2 Abstract Multicoordinated Paxos

As in [9], our proof of correctness starts with an abstract algorithm that can be more easily proved correct. The reason why we cannot use the same abstract algorithm as in [9] is the difficulty of multicoordinated rounds to implement the variable *minTried* used there. As a result, we came up with an even more abstract algorithm, which can be implemented not only by Multicoordinated Paxos, but also by the abstract algorithm in [9].

Our abstract algorithm is based upon a subset of the variables used by the abstract algorithm of [9]:

learned An array of c-structs, where $learned[l]$ is the c-struct currently learned by learner l. Initially, $learned[l] = \bot$ for all learners l.

propCmd The set of proposed commands. It initially equals the empty set.

bA A ballot array. It represents the current state of the voting. Initially, $\widehat{bA}_a = 0$, $bA_a[0] = \bot$ and $bA_a[m] = none$ for all $m > 0$. (Every acceptor casts a default vote for $\bot$ in ballot 0, so the algorithm begins with $\bot$ chosen.)

maxTried An array of c-structs, where $maxTried[m]$ is either a c-struct or equal to *none*, for every balnum m. Initially, $maxTried[0] = \bot$ and $maxTried[m] = none$ for all $m > 0$

The Abstract Multicoordinated Paxos algorithm satisfies the following invariants, which, as we prove next, imply the properties Nontriviality and Consistency of Generalized Consensus.

maxTried **Invariant** For every balnum m, if $maxTried[m] \neq none$, then

1. $maxTried[m]$ is proposed.
2. $maxTried[m]$ is safe at m in bA.

bA **Invariant** For all acceptors a and balnums m, if $bA_a[m] \neq none$, then

1. $bA_a[m]$ is safe at m in bA.
2. If m is a classic balnum, then $bA_a[m] \sqsubseteq maxTried[m]$.
3. If m is a fast balnum, then $bA_a[m]$ is proposed.

learned **Invariant** For every learner l:

1. $learned[l]$ is proposed.
2. $learned[l]$ is the lub of a finite set of c-structs chosen in bA.

Proposition 2 *The* learned *invariant implies the Nontriviality property of Generalized Consensus.*

PROOF: By part 1 of the *learned* invariant. □

Proposition 3 *Invariants* bA *and* learned *imply the Consistency property of Generalized Consensus.*

PROOF: By the definition of Consistency, it suffices to assume that invariants *bA* and *learned* are true, and prove that, for every pair of learners l_1 and l_2, $learned[l_1]$ and $learned[l_2]$ are compatible. The proof is divided into four steps, presented below:

1. bA is safe.
   PROOF: This follows from part 1 of the bA invariant and the definition of a safe ballot array (Definition 5).

LET: $S = \{v : v \text{ is chosen in } bA\}$

2. S is compatible.
   PROOF: By step 1 and Proposition 1.
3. For every learner l, $learned[l] \sqsubseteq \sqcup S$.
   PROOF: This is true by part 2 of the *learned* invariant and axiom CS3, which implies that if set S is compatible, then the lub of S is equal to or extends the lub of any subset of S.
4. Q.E.D.
   PROOF: By step 3 and the definition of compatible c-structs. □

Abstract Multicoordinated Paxos has seven atomic actions, described below. A complete specification of the algorithm in TLA$^+$ is given in Section B.2.

Propose(C) for any command C. It is enabled iff $C \notin propCmd$. It sets $propCmd$ to $propCmd \cup \{C\}$.

JoinBallot(a, m) for acceptor a and balnum m. It is enabled iff $\widehat{bA}_a < m$. It sets $\widehat{bA}_a$ to m.

StartBallot(m, w) for balnum m and c-struct w. It is enabled iff

- $maxTried[m] = none$,
- w is safe at m in bA, and
- $w \in Str(propCmd)$.

It sets $maxTried[m]$ to w.

Suggest(m, σ) for balnum m and c-seq σ. It is enabled iff

- $maxTried[m] \neq none$ and
- $\sigma \in Seq(propCmd)$.

It sets $maxTried[m]$ to $maxTried[m] \bullet \sigma$

ClassicVote(a, m, v) for acceptor a, balnum m, and c-struct v. It is enabled iff

- $m \geq \widehat{bA}_a$,

- v is safe at m in bA,
- $v \sqsubseteq maxTried[m]$, and
- $bA_a[m] = none$ or $bA_a[m] \sqsubseteq v$.

It sets $bA_a[m]$ to v and $\widehat{bA}_a$ to m.

FastVote(a, C) for acceptor a and command C. It is enabled iff

- $C \in propCmd$,
- $\widehat{bA}_a$ is a fast balnum, and
- $bA_a[\widehat{bA}_a] \neq none$.

It sets $bA_a[\widehat{bA}_a]$ to $bA_a[\widehat{bA}_a] \bullet C$.

AbstractLearn(l, v) for learner l and c-struct v. It is enabled iff v is chosen in bA. It sets $learned[l]$ to $learned[l] \sqcup v$.

The following proposition proves that the algorithm also satisfies the Stability property of Generalized Consensus.

Proposition 4 *Abstract Multicoordinated Paxos satisfies the Stability property of Generalized Consensus.*

PROOF: For any learner l, the only action that changes the value of $learned[l]$ is *AbstractLearn*(l, v). Since, by the definition of lub, this action can only extend the value of $learned[l]$, Stability is ensured. □

It remains to prove that the abstract algorithm satisfies the invariants *maxTried*, *bA*, and *learned*. For the sake of simplicity, however, we use some extra notation in the proof. First, when analyzing the execution of an action, we use ordinary expressions such as *exp* to represent the value of that expression before the action is executed, and we let exp' be the value of that expression after the action execution. Second, to avoid ambiguity, we let *maxTriedInv*, *bAInv*, and *learnedInv* be expressions representing the statements of the three invariant properties.

Proposition 5 *Abstract Multicoordinated Paxos satisfies the invariants* maxTried*,* bA*, and* learned*.*

PROOF: The invariants are trivially satisfied in the initial state. Therefore, it suffices to assume that the invariants are true and prove that, for every action α, they remain true if α is executed. We do that in the following, analyzing case by case.

1. CASE: Action $Propose(C)$ is executed, where $C \in Cmd$.

PROOF SKETCH: Action $Propose(C)$ only changes variable *propCmd*, which is the set of proposed values, and does that by adding a new element to it. Invariant conditions that do not refer to this set are obviously preserved. The others are kept true since the set *propCmd* only increases and c-structs composed of proposed commands remain composed of proposed values.

1.1. 1. $maxTried' = maxTried$
2. $bA' = bA$
3. $learned' = learned$
4. $propCmd' = propCmd \cup C \wedge C \notin propCmd$

PROOF: By the definition of action $Propose(C)$.

1.2. $maxTriedInv'$ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.

1.2.1. $maxTried[r]' \in Str(propCmd')$

PROOF: By applying the assumption of step 1.2 and step 1.1.1 to *maxTriedInv* we verify that $maxTried[r]' \in Str(propCmd)$, and step 1.1.4 tells us that $propCmd \subset propCmd'$.

1.2.2. $maxTried[r]'$ is safe at r in bA'.

PROOF: By *maxTriedInv* and steps 1.1.1 and 1.1.2.

1.2.3. Q.E.D.

1.3. $bAInv'$ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq none$, for any acceptor e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

1.3.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: By *bAInv* and step 1.1.2.

1.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: By *bAInv* and steps 1.1.1 and 1.1.2.

1.3.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$

PROOF: Step 1.1.2 and *bAInv* imply that, if r is a fast balnum, $bA_e[r]' \in propCmd$. Step 1.1.4 shows that $propCmd \subset propCmd'$.

1.3.4. Q.E.D.

1.4. $learnedInv'$ is true.

LET: h be any learner, without loss of generality.

1.4.1. $learned[h]' \in Str(propCmd')$

PROOF: Step 1.1.3 and *learnedInv* imply that $learned[h]' \in Str(propCmd)$, and step 1.1.4 implies that $propCmd \subset propCmd'$.

1.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: By *learnedInv* and steps 1.1.2 and 1.1.3.

1.4.3. Q.E.D.

1.5. Q.E.D.

2. CASE: Action $JoinBallot(a, m)$ is executed, where a is an acceptor and m is a ballot number.

PROOF SKETCH: Action $JoinBallot(a, m)$ only changes $\widehat{bA}_a$, setting it to m, which is bigger than $\widehat{bA}_a$. Invariant conditions that do not refer to $\widehat{bA}_a$ are obviously preserved. It remains to check the conditions stating that certain values are safe or chosen in bA'. The definition of *chosen* does not involve $\widehat{bA}_e$ for any acceptor e. The definition of *safe* is based upon the definition of *choosable at*, which does refer to $\widehat{bA}_e$, but implies that a value w that is choosable at round k in bA' is also choosable at k in bA. By the definition of *safe*, this implies that a value x that is safe at a balnum s in bA is also safe at s in bA'.

2.1. 1. $\widehat{bA}_a < m$

2. $\widehat{bA}'_a = m$

3. $\forall i \in Acceptor \setminus \{a\} : \widehat{bA}'_i = \widehat{bA}_i$

4. $\forall i \in Acceptor, j \in BalNum : bA_i[j]' = bA_i[j]$

5. $propCmd' = propCmd$

6. $maxTried' = maxTried$

7. $learned' = learned$

PROOF: By the definition of action $JoinBallot(a, m)$.

2.2. If x is safe at s in bA, for any c-struct x and balnum s, then x is safe at s in bA'.

The proof is by contradiction, as follows.

ASSUME: There exist c-struct x and balnum s, such that

1. x is safe at s in bA
2. x is not safe at s in bA'

PROVE: FALSE

2.2.1. Choose c-struct w and balnum k such that $k < s$, w is choosable at k in bA', and $w \not\sqsupseteq x$.

PROOF: w and k exist by assumption 2 of step 2.2 and the definition of *safe at*.

2.2.2. w is choosable at k in bA.

2.2.2.1. Choose k-quorum Q such that

$$\forall e \in Q : \widehat{bA}'_e > k \Rightarrow w \sqsubseteq bA_e[k]'$$

PROOF: Q exists by the definition of *choosable at*.

2.2.2.2. $\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k]$

PROOF: Steps 2.1.(1-3) imply that $\forall i \in Acceptor : \widehat{bA}_i > k \Rightarrow \widehat{bA}'_i > k$. Moreover, step 2.1.4 implies $\forall i \in Acceptor : bA_e[k] = bA_e[k]'$. If we combine this two formulas with the one of step 2.2.2.1, we can derive the expression of the current step.

2.2.2.3. Q.E.D.

PROOF: By step 2.2.2.2 and the definition of *choosable at*.

2.2.3. Q.E.D.

PROOF: If w is choosable at $k < s$ in bA (step 2.2.2), and $w \not\sqsubseteq x$ (step 2.2.1), then x is not safe at s in bA, contradicting assumption 1 of step 2.2.

2.3. $maxTriedInv'$ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.

2.3.1. $maxTried[r]' \in Str(propCmd')$

PROOF: By *maxTriedInv* and steps 2.1.5 and 2.1.6.

2.3.2. $maxTried[r]'$ is safe at r in bA'.

PROOF: By *maxTriedInv*, step 2.1.6, and step 2.2.

2.3.3. Q.E.D.

2.4. $bAInv'$ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq none$, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

2.4.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: By *bAInv*, step 2.1.4, and step 2.2.

2.4.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: By *bAInv* and steps 2.1.4 and 2.1.6.

2.4.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$

PROOF: By *bAInv* and steps 2.1.4 and 2.1.5.

2.4.4. Q.E.D.

2.5. $learnedInv'$ is true.

LET: h be any learner, without loss of generality.

2.5.1. $learned[h]' \in Str(propCmd')$

PROOF: By *learnedInv* and steps 2.1.5 and 2.1.7.

2.5.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.
PROOF: *learnedInv* and step 2.1.7 imply that $learned[h]'$ is the lub of a finite set of c-structs chosen in bA. Step 2.1.4 and the definition of a chosen value imply that a value that is chosen in bA is also chosen in bA'.
2.5.3. Q.E.D.
2.6. Q.E.D.

3. CASE: Action $StartBallot(m, w)$ is executed, where m is a balnum and $w \in CStruct$.

PROOF SKETCH: Action $StartBallot(m, w)$ changes $maxTried[m]$ from *none* to w, which is ensured to be both proposed and safe at m in bA. The action does not change the other variables. It preserves the *maxTried* invariant because w is proposed and safe at m in bA. It preserves the bA invariant because it does not change bA and $bA_e[m]$ is ensured to equal *none*, for any acceptor e, by the bA invariant itself. It preserves the *learned* invariant because it does not change *learned* or bA.

3.1. 1. $maxTried[m] = none$
2. w is safe at m in bA
3. $w \in Str(propCmd)$
4. $propCmd' = propCmd$
5. $maxTried[m]' = w$
6. $\forall i \in BalNum \setminus \{m\} : maxTried[i]' = maxTried[i]$
7. $bA' = bA$
8. $learned' = learned$

PROOF: By the definition of action $StartBallot(m, w)$.
3.2. $maxTriedInv'$ is true.
From its definition, it suffices to:
ASSUME: $maxTried[r]' \neq none$, for any balnum r
PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.
3.2.1. $maxTried[r]' \in Str(propCmd')$
PROOF: If $r = m$, by steps 3.1.(3-5). If $r \neq m$, it is implied by *maxTriedInv* and steps 3.1.4 and 3.1.6.
3.2.2. $maxTried[r]'$ is safe at r in bA'.
PROOF: If $r = m$, by steps 3.1.2, 3.1.5, and 3.1.7. If $r \neq m$, it is implied by *maxTriedInv* and steps 3.1.(6-7).
3.2.3. Q.E.D.
3.3. $bAInv'$ is true.
From its definition, it suffices to:
ASSUME: $bA_e[r]' \neq none$, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

3.3.1. $bA_e[r]'$ is safe at r in bA'.
PROOF: By *bAInv* and step 3.1.7.

3.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$
PROOF: Step 3.1.1 and *bAInv* imply that $bA_e[m] = none$. Now, if we apply step 3.1.7, we get that $bA_e[m]' = none$. This implies, by the assumption of step 3.3, that $r \neq m$. Thus, the proof follows from *bAInv* and steps 3.1.(6-7).

3.3.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$
PROOF: By *bAInv* and steps 3.1.4 and 3.1.7.

3.3.4. Q.E.D.

3.4. *learnedInv*$'$ is true.
LET: h be any learner, without loss of generality.

3.4.1. $learned[h]' \in Str(propCmd')$
PROOF: By *learnedInv* and steps 3.1.4 and 3.1.8.

3.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.
PROOF: By *learnedInv* and steps 3.1.(7-8).

3.4.3. Q.E.D.

3.5. Q.E.D.

4. CASE: Action $Suggest(m, \sigma)$ is executed, where m is a ballot number and σ is a c-seq.

PROOF SKETCH: Action $Suggest(m, \sigma)$ changes $maxTried[m]$ to $maxTried[m] \bullet \sigma$, where σ is a sequence of proposed commands. The action does not change the other variables. It preserves the *maxTried* invariant because σ is proposed and, by the definition of *safe at*, any extension of a value that is safe at m in bA is also safe at m in bA. It preserves the bA invariant because it does not change bA and the bA invariant ensures that $bA_e[m]$ is either *none* or a value v such that $v \sqsubseteq maxTried[m]$, for any acceptor e. It preserves the *learned* invariant because it does not change *learned* or bA.

4.1. 1. $maxTried[m] \neq none$
2. $\sigma \in Seq(propCmd)$
3. $propCmd' = propCmd$
4. $maxTried[m]' = maxTried[m] \bullet \sigma$
5. $\forall i \in BalNum \setminus \{m\} : maxTried[i]' = maxTried[i]$
6. $bA' = bA$
7. $learned' = learned$

PROOF: By the definition of action $Suggest(m, \sigma)$.

4.2. *maxTriedInv*$'$ is true.
From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r
PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.

4.2.1. $maxTried[r]' \in Str(propCmd')$
PROOF: If $r = m$, by *maxTriedInv* and steps 4.1.(2-4). If $r \neq m$, it is implied by *maxTriedInv* and steps 4.1.3 and 4.1.5.

4.2.2. $maxTried[r]'$ is safe at r in bA'.
PROOF: *maxTriedInv* implies that $maxTried[r]$ is safe at r in bA, and step 4.1.6 states that $bA = bA'$. Steps 4.1.(4-5) complete the proof by implying that $maxTried[r] \sqsubseteq maxTried[r]'$. This is enough because the definition of *safe at* implies that if w is safe at k in β, then v is safe at k in β, for any v such that $w \sqsubseteq v$.

4.2.3. Q.E.D.

4.3. $bAInv'$ is true.
From its definition, it suffices to:
ASSUME: $bA_e[r]' \neq none$, for any agent e and balnum r
PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

4.3.1. $bA_e[r]'$ is safe at r in bA'.
PROOF: By *bAInv* and step 4.1.6.

4.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$
PROOF: Step 4.1.6 implies that $bA_e[r]' = bA_e[r]$, *bAInv* implies that $bA_e[r] \sqsubseteq maxTried[r]$, and steps 4.1.(4-5) imply that $maxTried[r] \sqsubseteq maxTried[r]'$, completing the proof.

4.3.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$
PROOF: By *bAInv* and steps 4.1.3 and 4.1.6.

4.3.4. Q.E.D.

4.4. *learnedInv*$'$ is true.
LET: h be any learner, without loss of generality.

4.4.1. $learned[h]' \in Str(propCmd')$
PROOF: By *learnedInv* and steps 4.1.3 and 4.1.7.

4.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.
PROOF: By *learnedInv* and steps 4.1.(6-7).

4.4.3. Q.E.D.

4.5. Q.E.D.

5. CASE: Action $ClassicVote(a, m, v)$ is executed, where a is an acceptor, m is a ballot number, and $v \in CStruct$.

PROOF SKETCH: Action *ClassicVote*(a, m, v) sets $bA_a[m]$ to c-struct v only if $m \geq \widehat{bA}_a$, v is ensured to be safe at m in bA, and $bA_a[m]$ equals *none* or $bA_a[m] \sqsubseteq v$. It is also a pre-condition to this action that $v \sqsubseteq maxTried[m]$, which implies that v is proposed, by the *maxTried* invariant. Since only entry $bA_a[m]$ ($m \geq \widehat{bA}_a$) is changed together with $\widehat{bA}_a$, which is set to m, and the definition of *choosable at* considers only entries $bA_e[j]$ where $j < \widehat{bA}_e$, no value can be made unsafe at any balnum after the execution of this action. It preserves the *maxTried* invariant because it does not change *maxTried* or *propCmd* and it does not make any entry unsafe. It preserves the bA invariant because no entry is made unsafe and the only entry it changes in bA is set to a safe and proposed value. It preserves the *learned* invariant because it does not change *learned* and any value that is chosen in bA remains chosen after the action is executed, by the definition of *chosen*.

5.1. 1. $m \geq \widehat{bA}_a$
2. v is safe at m in bA
3. $v \sqsubseteq maxTried[m]$
4. $bA_a[m] = none \vee bA_a[m] \sqsubseteq v$
5. $propCmd' = propCmd$
6. $maxTried' = maxTried$
7. $\forall i \in Acceptor \setminus \{a\} : \widehat{bA}_i' = \widehat{bA}_i$
8. $\widehat{bA}_a' = m$
9. $\forall i \in Acceptor, j \in BalNum : (i \neq a \vee j \neq m) \Rightarrow bA_i[j]' = bA_i[j]$
10. $bA_a[m]' = v$
11. $learned' = learned$

PROOF: By the definition of action *ClassicVote*(a, v).

5.2. If x is safe at s in bA, for any c-struct x and balnum s, then x is safe at s in bA'.

The proof is by contradiction, as follows.

ASSUME: There exist c-struct x and balnum s, such that
1. x is safe at s in bA
2. x is not safe at s in bA'

PROVE: FALSE

5.2.1. Choose c-struct w and balnum k such that $k < s$, w is choosable at k in bA', and $w \not\sqsubseteq x$.

PROOF: w and k exist by assumption 5.2.2 and the definition of *safe at*.

5.2.2. w is choosable at k in bA.

5.2.2.1. Choose k-quorum Q such that
$$\forall e \in Q : \widehat{bA}_e' > k \Rightarrow w \sqsubseteq bA_e[k]'$$
PROOF: Q exists by the definition of *choosable at*.

5.2.2.2. $\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k]$

PROOF: Steps 5.1.(8-9) imply that $\forall i \in Acceptor : \widehat{bA}_i' > k \Rightarrow$

$bA_i[k]' = bA_i[k]$. This equation applied to the one of step 5.2.2.1 leads to

$$\forall e \in Q : \widehat{bA}_e' > k \Rightarrow w \sqsubseteq bA_e[k].$$

Steps 5.1.(1,7-8) imply that $\forall i \in Acceptor : \widehat{bA}_i' \geq \widehat{bA}_i$. Thus, $\forall e \in Q : \widehat{bA}_e > k \Rightarrow \widehat{bA}_e' > k$, which together with the previous equation leads us to

$$\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k].$$

5.2.2.3. Q.E.D.

PROOF: By step 5.2.2.2 and the definition of *choosable at*.

5.2.3. Q.E.D.

PROOF: If w is choosable at $k < s$ in bA (step 5.2.2), and $w \not\sqsubseteq x$ (step 5.2.1), then x is not safe at s in bA, contradicting assumption 1 of step 5.2.

5.3. $maxTriedInv'$ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.

5.3.1. $maxTried[r]' \in Str(propCmd')$

PROOF: By steps 5.1.(5-6).

5.3.2. $maxTried[r]'$ is safe at r in bA'.

PROOF: By $maxTriedInv$, step 5.1.6 and step 5.2.

5.3.3. Q.E.D.

5.4. $bAInv'$ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq none$, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

5.4.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: If $e = a$ and $r = m$, it follows from steps 5.1.2 and 5.1.10. Otherwise, it follows from $bAInv$, step 5.1.9, and step 5.2.

5.4.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: If $e = a$ and $r = m$, it follows from steps 5.1.3 and 5.1.10. Otherwise, it follows from $bAInv$ and step 5.1.9.

5.4.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$

PROOF: If $e = a$ and $r = m$, it follows from steps 5.1.3 and 5.1.10, and $maxTriedInv$. Otherwise, it follows from $bAInv$ and step 5.1.9.

5.4.4. Q.E.D.

5.5. *learnedInv*′ is true.

LET: h be any learner, without loss of generality.

5.5.1. $learned[h]' \in Str(propCmd')$

PROOF: By *learnedInv* and steps 5.1.5 and 5.1.11.

5.5.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: *learnedInv* and step 5.1.11 imply that $learned[h]'$ is the lub of a finite set of c-structs chosen in bA. Steps 5.1.(4,9-10) state that the only entry of bA that is modified, is extended. The definition of a chosen value, therefore, implies that a value that is chosen in bA is also chosen in bA', completing the proof.

5.5.3. Q.E.D.

5.6. Q.E.D.

6. CASE: Action $FastVote(a, C)$ is executed, where a is an acceptor and $C \in Cmd$.

PROOF SKETCH: Action $FastVote(a, C)$ sets $bA_a[\widehat{bA}_a]$ to c-struct $bA_a[\widehat{bA}_a] \bullet C$ only if $bA_a[\widehat{bA}_a]$ does not equal *none* and C is proposed. Since only $bA_a[\widehat{bA}_a]$ is changed and the definition of *choosable at* considers only entries $bA_e[m]$ where $m < \widehat{bA}_e$, no value can be made unsafe at any balnum after the execution of this action. It preserves the *maxTried* invariant because it does not change *maxTried* or *propCmd* and it does not make any entry unsafe. It preserves the bA invariant because no entry is made unsafe and the only entry it changes in bA is extended with a proposed value, and the extension of a safe c-struct is also safe. It preserves the *learned* invariant because it does not change *learned* and any value that is chosen in bA remains chosen after the action is executed, by the definition of *chosen*.

6.1. 1. $C \in propCmd$

2. $\widehat{bA}_a$ is a fast balnum

3. $bA_a[\widehat{bA}_a] \neq none$

4. $propCmd' = propCmd$

5. $maxTried' = maxTried$

6. $bA_a[\widehat{bA}_a]' = bA_a[\widehat{bA}_a] \bullet C$

7. $\forall i \in Acceptor : \widehat{bA}'_i = \widehat{bA}_i$

8. $\forall i \in Acceptor, j \in BalNum : (i \neq a \vee j \neq \widehat{bA}_a) \Rightarrow bA_i[j]' = bA_i[j]$

9. $learned' = learned$

PROOF: By the definition of action $FastVote(a, v)$.

6.2. If x is safe at s in bA, for any c-struct x and balnum s, then x is safe at s in bA'.

The proof is by contradiction, as follows.

ASSUME: There exist c-struct x and balnum s, such that

1. x is safe at s in bA
2. x is not safe at s in bA'

PROVE: FALSE

6.2.1. Choose c-struct w and balnum k such that $k < s$, w is choosable at k in bA', and $w \not\sqsubseteq x$.

PROOF: w and k exist by assumption 6.2.2 and the definition of *safe at*.

6.2.2. w is choosable at k in bA.

6.2.2.1. Choose k-quorum Q such that

$$\forall e \in Q : \widehat{bA}'_e > k \Rightarrow w \sqsubseteq bA_e[k]'$$

PROOF: Q exists by the definition of *choosable at*.

6.2.2.2. $\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k]$

PROOF: Step 6.1.7 states that $\widehat{bA}'_e = \widehat{bA}_e$, and step 6.1.8 implies that $bA_e[k]' = bA_e[k]$ if $\widehat{bA}'_e > k$.

6.2.2.3. Q.E.D.

PROOF: By step 6.2.2.2 and the definition of *choosable at*.

6.2.3. Q.E.D.

PROOF: If w is choosable at $k < s$ in bA (step 6.2.2), and $w \not\sqsubseteq x$ (step 6.2.1), then x is not safe at s in bA, contradicting assumption 1 of step 6.2.

6.3. *maxTriedInv*$'$ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.

6.3.1. $maxTried[r]' \in Str(propCmd')$

PROOF: By steps 6.1.(4-5).

6.3.2. $maxTried[r]'$ is safe at r in bA'.

PROOF: By *maxTriedInv*, step 6.1.5 and step 6.2.

6.3.3. Q.E.D.

6.4. *bAInv*$'$ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq none$, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

6.4.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: If $e = a$ and $r = \widehat{bA}_a$, it follows from steps 6.1.3 and 6.1.6, *bAInv*, and the definition of *safe at*, which implies that the extension of a safe value is also safe. Otherwise, it follows from *bAInv*, step 6.1.8, and step 6.2.

6.4.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: It follows from *bAInv* and step 6.1.8, since $r = \widehat{bA}_e$ and $e = a$ imply that r is not classic, by step 6.1.2.

6.4.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$

PROOF: If $e = a$ and $r = \widehat{bA}_a$, it follows from *bAInv* and steps 6.1.1 and 6.1.6. Otherwise, it follows from *bAInv* and step 6.1.8.

6.4.4. Q.E.D.

6.5. *learnedInv*$'$ is true.

LET: h be any learner, without loss of generality.

6.5.1. $learned[h]' \in Str(propCmd')$

PROOF: By *learnedInv* and steps 6.1.4 and 6.1.9.

6.5.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: *learnedInv* and step 6.1.9 imply that $learned[h]'$ is the lub of a finite set of c-structs chosen in bA. Steps 6.1.(3,6-8) state that the only entry of bA that is modified, is extended. The definition of a chosen value, therefore, implies that a value that is chosen in bA is also chosen in bA', completing the proof.

6.5.3. Q.E.D.

6.6. Q.E.D.

7. CASE: Action $AbstractLearn(l, v)$ is executed, where l is a learner and $v \in CStruct$.

PROOF SKETCH: Action $AbstractLearn(l, v)$ only changes variable *learned*, which is the array of learned c-structs, and does that by extending one entry to the lub of it with a chosen c-struct. Invariants *maxTried* and *bA* are obviously preserved. The first part of the *learned* invariant is preserved because this extension is proposed, by the definition of *chosen at*, the *bA* invariant and axiom CS3. The second part is obviously preserved by its definition and the one of the action.

7.1. 1. v is chosen in bA

2. $propCmd' = propCmd$

3. $maxTried' = maxTried$

4. $bA' = bA$

5. $learned[l]' = learned[l] \sqcup v$

6. $\forall i \in Learner \setminus \{l\} : learned[i]' = learned[i]$

PROOF: By the definition of action $AbstractLearn(l, v)$.

7.2. *maxTriedInv*$'$ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and

2. $maxTried[r]'$ is safe at r in bA'.

7.2.1. $maxTried[r]' \in Str(propCmd')$
PROOF: By *maxTriedInv* and steps 7.1.(2-3).
7.2.2. $maxTried[r]'$ is safe at r in bA'.
PROOF: By *maxTriedInv* and steps 7.1.(3-4).
7.2.3. Q.E.D.
7.3. $bAInv'$ is true.
From its definition, it suffices to:
ASSUME: $bA_e[r]' \neq none$, for any acceptor e and balnum r
PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.
7.3.1. $bA_e[r]'$ is safe at r in bA'.
PROOF: By *bAInv* and step 7.1.4.
7.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$
PROOF: By *bAInv* and steps 7.1.(3-4).
7.3.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$
PROOF: By *bAInv* and steps 7.1.2 and 7.1.4.
7.3.4. Q.E.D.
7.4. *learnedInv*$'$ is true.
LET: h be any learner, without loss of generality.
7.4.1. $learned[h]' \in Str(propCmd')$
PROOF: If $h \neq l$, it follows from *learnedInv* and steps 7.1.6 and 7.1.2. Otherwise, the definition of a chosen value, *bAInv*, and step 7.1.1 imply that $v \in Str(propCmd)$. *learnedInv* implies that $learned[l] \in Str(propCmd)$. Proposition 1 and *learnedInv* imply that v and $learned[l]$ are compatible, and axiom CS3 states that its lub exists and must be in $Str(propCmd)$. The proof is completed by steps 7.1.2 and 7.1.5.
7.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.
PROOF: If $h \neq l$, it follows from *learnedInv* and step 7.1.6. Otherwise, it follows from *learnedInv* and steps 7.1.1 and 7.1.5.
7.4.3. Q.E.D.
7.5. Q.E.D.

□

A.3 Distributed Abstract Multicoordinated Paxos

As an intermediate step in our proof, we introduce a distributed version of the abstract algorithm in the previous section. This algorithm has the vari-

ables *propCmd*, *learned*, and *bA* with the same role as in the non-distributed abstract algorithm. It introduces the variables *dMaxTried*, a distributed version of *maxTried*, and *msgs*, used to simulate a message passing system by holding the messages sent between coordinators, acceptors, and learners.

propCmd The set of proposed commands. It initially equals the empty set.

learned An array of c-structs, where $learned[l]$ is the c-struct currently learned by learner l. Initially, $learned[l] = \bot$ for all learners l.

bA A ballot array. It represents the current state of the voting. Initially, $\widehat{bA}_a = 0$, $bA_a[0] = \bot$ and $bA_a[m] = none$ for all acceptor a and ballot number $m > 0$. (Every acceptor casts a default vote for $\bot$ in ballot 0, so the algorithm begins with $\bot$ chosen.)

dMaxTried An array of arrays of c-structs, where $dMaxTried[c][m]$ is either a c-struct or equal to *none*, for every coordinator c and balnum m. Initially, $dMaxTried[c][0] = \bot$ and $dMaxTried[c][m] = none$ for every coordinator c and all balnum $m > 0$.

msgs The set of messages sent by coordinators and acceptors. (This set is used to simulate the message passing among processes.)

The distributed abstract algorithm is described in terms of the following actions. Its formal specification in TLA$^+$ is given in the appendix section B.3.

Propose(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to $propCmd \cup \{C\}$, from where coordinators and acceptors can read C.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff $dMaxTried[c][m] = none$. It sends the message $\langle \text{“}\texttt{1a}\text{”}, m\rangle$ to acceptors (adds it to *msgs*).

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

- $\widehat{bA}_a < m$
- $\langle \text{“}\texttt{1a}\text{”}, m\rangle \in msgs$

It sets $\widehat{bA}_a$ to m and sends the message $\langle \text{“}\texttt{1b}\text{”}, m, bA_a\rangle$ to the coordinators.

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff:

- $dMaxTried[c][m] = none$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle \text{“}\texttt{1b}\text{”}, m, bA_a \rangle \in msgs$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, m, \beta)$, and β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = m$ and c has received a message $\langle \text{“}\texttt{1b}\text{”}, m, \rho \rangle$ from a with $\rho = \beta_a$.

This action sets $dMaxTried[c][m]$ to v and sends the message $\langle \text{“}\texttt{2a}\text{”}, m, v \rangle$ to acceptors.

Phase2aClassic(c, m, C) executed by coordinator c, for balnum m and command C. The action is enabled iff

- $C \in propCmd$.
- $dMaxTried[c][m] \neq none$

This action sends the message $\langle \text{“}\texttt{2a}\text{”}, m, c, dMaxTried[c][m] \bullet C \rangle$ to the acceptors and sets $dMaxTried[c][m]$ to $dMaxTried[c][m] \bullet C$.

Phase2bClassic(a, m, v) executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq \widehat{bA}_a$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase “2a” message for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $bA_a[m]$ equals $none$ and v equals u, or $bA_a[m]$ and u are compatible and v equals $bA_a[m] \sqcup u$.

It sets $bA_a[m]$ to v, $\widehat{bA}_a$ to m, and sends the message $\langle \text{“}\texttt{2b}\text{”}, m, v \rangle$ to the learners.

Phase2bFast(a, C) executed by acceptor a, for balnum m and command C. The action is enabled iff

- $\widehat{bA}_a$ is a fast balnum,
- $bA_a[\widehat{bA}_a] \neq none$, and

- $C \in propCmd$.

It sets $bA_a[\widehat{bA}_a]$ to $bA_a[\widehat{bA}_a] \bullet C$ and sends a message $\langle$"2b", $\widehat{bA}_a$, $bA_a[\widehat{bA}_a] \bullet C\rangle$ to the learners.

Learn(l, v) executed by learner l, for c-struct v. It is enabled iff a has received phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

The distributed abstract algorithm implements the the non-distributed version in the sense that all behaviors of the former are also behaviors of the latter. This implementation is stated by the following proposition.

Proposition 6 *Distributed Abstract Multicoordinated Paxos implements the Abstract Multicoordinated Paxos specification.*

To prove this proposition we give a refinement mapping [1] from the distributed version's states to the non-distributed version's.

In the following we replace the variables in the non-distributed algorithms by overlined versions. That is, we let expression $\overline{A}$ refer to expression E, in the non-distributed algorithm specification, where all occurrences of variables *propCmd*, *maxTried*, bA, and *learned* are replaced by $\overline{propCmd}$, $\overline{maxTried}$, $\overline{bA}$, and $\overline{learned}$, respectively. Non-overlined expressions refer to those in the distributed algorithm. We give the refinement mapping by defining overlined variables based on the distributed algorithm's variables in a way that satisfies the non-distributed specification.

Let the overlined variables be defined as follows.

$$\overline{propCmd} \triangleq propCmd$$

$$\overline{learned} \triangleq learned$$

$$\begin{array}{l}
\overline{maxTried} \triangleq \\
\quad \text{LET } Tried(Q, m) \triangleq \text{IF } \exists c \in Q : dMaxTried[c][m] = none \\
\qquad\qquad\qquad\qquad \text{THEN } none \\
\qquad\qquad\qquad\qquad \text{ELSE } \sqcap \{dMaxTried[c][m] : c \in Q\} \\
\qquad AllTried(m) \triangleq \{v \in \{Tried(Q, m) : Q \text{ is an } m\text{-coordquorum}\} : \\
\qquad\qquad\qquad\qquad v \neq none\} \\
\quad \text{IN } \quad [m \in BalNum \mapsto \text{IF } AllTried(m) = \{\} \text{ THEN } none \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } \sqcup AllTried(m)]
\end{array}$$

To prove that this is a valid refinement mapping and witnesses the implementation of the Abstract Multicoordinated Paxos by Distributed Abstract Multicoordinated Paxos we must show that the distributed version's intial states imply the non-distributed version's initial states, and that each step of the distributed version implies a step in the non-distributed version, be it one that changes the overlined variables or does not change anything (a stuttering step). To simplify these proofs we first prove some properties of c-structs and of our refinement mapping. (The first proposition actually regards c-structs in general)

Proposition 7 *If, for some ballot number m and ballot array bA, all elements of S are safe at m in bA, then $\sqcup S$ is also safe at m in bA.*

By the definition of "safe at", for all c-structs v choseable at round m in bA and for all $w \in S$, $v \sqsubseteq w$. Moreover, by the definition of "lower bound", v is a lower bound of S and, by the definition of $\sqcup$, $v \sqsubseteq \sqcup S$. Therefore, all elements of S extend v and all c-structs choosable at m in bA, being safe at m in bA. □

Proposition 8 *$AllTried(m)$ is compatible for $m \geq 0$.*

Proof: By the definition of glb, the empty set is compatible and $\sqcup\{\} = \bot$. If $AllTried(m)$ is not empty, then it contains the $Tried(Q, m)$ for all Q such that $Tried(Q, m) \neq none$. By the definition of $Tried$ and $\sqcap$, if $Tried(Q, m) \neq none$, then it is a prefix of $dMaxTried[c][m]$ for all $c \in Q$. Let $Tried(Q, m)$, $Tried(R, m) \in AllTried(m)$. By the coord-quorum-assumption, there exists a coordinator $c \in Q \cap R$, and $AllTried(Q, m) \sqsubseteq dMaxTried[c][m]$ and $AllTried(R, m) \sqsubseteq dMaxTried[c][m]$. Therefore, $dMaxTried[c][m]$ is an upper bound to $\{Tried(Q, m), Tried(R, m)\}$, and they are compatible. Because its elements are pairwise compatible and due to **CS3**, $AllTried(m)$ is compatible. □

Proposition 9 *$dMaxTried = dmaxTried' \Rightarrow \overline{maxTried} = \overline{maxTried}'$.*

Proof: $\overline{maxTried}$ is defined only over $dMaxTried$ values. If $dMaxTried$ does not change, then $maxTried$ cannot change. □

Hereinafter we refer to to Distributed Abstract Multicoordinated Paxos as DAP and Abstract Multicoordinated Paxos as AP. As we mentioned before, the proof of Proposition 6, i.e., that DAP implements the AP, is divided in two steps: proving the implication among the initial states and among the steps. The first step is captured by Proposition 10 and the second step by Proposition 11, below.

Proposition 10 *DAP's initial state implies AP's initial state.*

PROOF SKETCH: First we prove that DAP's initial state implies that $\overline{maxTried}$ is initialized as specified in AP's. Because the correct initialization of the other variables are trivially implied (they have the same initialization), we conclude the proof.

1. $\overline{maxTried}[0] = \bot$
   PROOF: By the specification of DAP, $dMaxTried[c][0] = \bot$ for each coordinator c. By the definition of $\overline{maxTried}$, $Tried$, and glb, $Tried(Q, 0) = \bot$ for any Q. Therefore, by definition of $AllTried$, $AllTried(0) = \bot$, and by the definition of lub, $\overline{maxTried}[0] = \bot$.
2. $\forall m > 0, \overline{maxTried}[m] = none$
   PROOF: By the specification of DAP, $dMaxTried[c][m] = none$ for each coordinator c and round $m > 0$. By the definition of $\overline{maxTried}$ and $Tried$, $Tried(Q, m) = none$ for any Q. Hence, by the definition of $AllTried$, $AllTried(m) = \{\}$. By the definition of $\overline{maxTried}$, $\overline{maxTried}[m] = none$ for any $m > 0$.
3. Q.E.D.

□

Proposition 11 *A DAP step implements an AP step (a possibly stuttering one).*

PROOF SKETCH: We consider each action of DAP and show that each step either implies a step of AP or that AP's variables—propCmd, learned, bA, and $\overline{maxTried}$—are left unchanged. We use TLA$^+$ UNCHANGED v notation to indicate that variable (or sequence of variables) v did not change in some step.

1. ASSUME: $\land\ C \in Cmd$
   $\land\ Propose(C)$
   PROVE: $\overline{Propose(C)}$
   1.1. $C \notin propCmd$
   PROOF: By the definition of $Propose$.
   1.2. $propCmd' = propCmd \cup \{C\}$
   PROOF: By the definition of $Propose$.
   1.3. UNCHANGED $\langle learned, bA, \overline{maxTried}\rangle$
   PROOF: By the definition of $Propose$ and Proposition 9.
   1.4. Q.E.D.

2. ASSUME: $\land\ c \in Coord$
   $\land\ m \in BalNum$
   $\land\ Phase1a(c, m)$
   PROVE: UNCHANGED $\langle propCmd, learned, bA, \overline{maxTried}\rangle$

PROOF: By the definition of *Phase1a* and Proposition 9.

3. ASSUME: $\land\ c \in Coord$
$\land\ m \in BalNum$
$\land\ v \in CStruct$
$\land\ Phase2Start(c, m, v)$
PROVE: $\land \lor \land\ \overline{maxTried} = none$
$\land \lor\ \overline{maxTried}' = none$
$\lor\ StartBallot(m, \overline{maxTried}'[m])$
$\lor \land\ \overline{maxTried} \neq none$
$\land \lor \exists \sigma \in Seq(propCmd) :$
$\land\ \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
$\land\ Suggest(m, \sigma)$
$\lor$ UNCHANGED $\overline{maxTried}$
$\land$ UNCHANGED $\langle propCmd, bA, learned \rangle$

3.1. ASSUME: $\overline{maxTried}[m] = none$
PROVE: $\lor\ StartBallot(m, \overline{maxTried}'[m])$
$\lor$ UNCHANGED $\overline{maxTried}$

3.1.1. ASSUME: $\overline{maxTried}'[m] \neq none$
PROVE: $StartBallot(m, \overline{maxTried}'[m])$

3.1.1.1. $\overline{maxTried}'[m] \sqsubseteq v$
PROOF: By the assumption, for all m-coordquorums Q such that $Tried(Q, m)' \in AllTried(m)'$, $c \in Q$, or $Tried(Q, m)'$ would equal *none* and it would not belong to $AllTried(m)'$. By the definition of glb, $Tried(Q, m)' \sqsubseteq dMaxTried[c][m] = v$. Therefore, all elements of $AllTried(m)'$ are compatible prefixes of v, and v is an upper bound of $AllTried(m)'$. But, by the definition of $\overline{maxTried}$, $\overline{maxTried}'[m] = \sqcup AllTried(m)'$ and by the definition of $\sqcup$ and Assumption CS3, $\overline{maxTried}'[m]$ must be a prefix of v.

3.1.1.2. $\overline{maxTried}[m] = none$
PROOF: By assumption.

3.1.1.3. $\overline{maxTried}'[m]$ is safe at m in bA
PROOF: By the definitions of actions *ProvedSafe*, *Phase2Start*, and *Phase2aClassic*, for all $d \in Coord$, $dMaxTried[d][m]$ is first set to a safe value and then simply extended, therefore remaining safe. By the definition of *AllTried* and Proposition 7, all elements of $AllTried(m)'$ are safe. Because $\overline{maxTried}[m]$ is an extension of such safe c-structs, it is also safe.

3.1.1.4. $\overline{maxTried}'[m] \in Str(propCmd)$

PROOF: By the definition of action $Phase2Start$, $v \in Str(propCmd)$. But, by step 3.1.1.1, $\overline{maxTried}'[m] \sqsubseteq v$ and $\overline{maxTried}'[m]$ is constructible from a subset of the commands in v. Therefore, by the definition of Str, $\overline{maxTried}'[m] \in Str(propCmd)$.

3.1.1.5. Q.E.D.

3.1.2. ASSUME: $\overline{maxTried}'[m] = none$
PROVE: UNCHANGED $\overline{maxTried}$

PROOF: By assumption.

3.1.3. Q.E.D.

3.2. ASSUME: $\overline{maxTried}[m] \neq none$
PROVE: $\lor \exists \sigma \in Seq(propCmd) :$
$\land\ \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
$\land\ Suggest(m, \sigma)$
$\lor$ UNCHANGED $\overline{maxTried}$

3.2.1. ASSUME: $\overline{maxTried}'[m] \neq \overline{maxTried}[m]$
PROVE: $\exists \sigma \in Seq(propCmd) :$
$\land\ \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
$\land\ Suggest(m, \sigma)$

3.2.1.1. $\exists \sigma \in Seq(propCmd) : \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$

PROOF: Let QD be the set of m-coordquorums Q such that $Tried(Q, m) \neq Tried(Q, m)'$; clearly by the assumption, QD is not empty and for all $Q \in QD$, $c \in Q$. Moreover, by the definition of $Tried$, $Tried(Q, m)' \sqsubseteq v$.

Let $SomeTried(SD, m) = \sqcup\{AllTried(S, m) : S \in SD\}$. Because for all $Q \in QD$, $Tried(Q, m) = none$, $SomeTried(\{S : S$ is an m-coordquorum and $S \notin QD\}, m) = AllTried(m)$.

Let Q and R be two m-coordquorums such that $Q \in QD$, $R \notin QD$, and $Q \cap R \neq \emptyset$; let $d \in Q \cap R$. So $Tried(R, m) = Tried(R, m)' \sqsubseteq dMaxTried[d][m]$ and $Tried(Q, m)' \sqsubseteq dMaxTried[d][m]$, and either $Tried(Q, m)' \sqsubseteq Tried(R, m)$ or $Tried(R, m) \sqsubseteq Tried(Q, m)'$. In the first case, $SomeTried(\{S : S$ is an m-coordquorum and $S \notin QD\}, m) = SomeTried(\{S : S$ is an m-coordquorum and $S \notin QD\} \cup \{Q\}, m)$, and is of no interest. In the second case, the equality may not hold, what would imply that $Tried(Q, m)'$ has some command that not in $AllTried(m)$. This second case must hold for some pair Q, R, since by assumption and the definition of $\overline{maxTried}$ $AllTried(m) \neq AllTried(m)'$. Without loss of generality, let R be such that $Tried(R, m)$ is maximal; by the definition of $\sqsubseteq$, $Tried(Q, m)' = Tried(R, m) \bullet \sigma$, for some sequence σ. As

$Tried(Q, m)' \sqsubseteq v \in Seq(propCmd)$, $\sigma \in Seq(propCmd)$. Finally, by the definition of $\sqcup$, $AllTried(m)' = AllTried(m)$.

3.2.1.2. ASSUME: $\exists \sigma \in Seq(propCmd) :$
$\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
PROVE: $Suggest(m, \sigma)$

PROOF: All pre and post-conditions of *Suggest* are assumed:

- $maxTried[m] \neq none$,
- $\sigma \in Seq(propCmd)$, and
- $\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$.

3.2.1.3. Q.E.D.

3.2.2. ASSUME: $\overline{maxTried}'[m] = \overline{maxTried}[m]$
PROVE: UNCHANGED $\overline{maxTried}$

PROOF: By the assumption.

3.2.3. Q.E.D.

3.3. UNCHANGED $\langle propCmd, bA, learned \rangle$

PROOF: This is trivially true, because variables *propCmd*, *bA*, and *learned* are kept unchanged in *Phase2Start*.

3.4. Q.E.D.

4. ASSUME: $\land\ c \in Coord$
$\land\ m \in BalNum$
$\land\ C \in propCmd$
$\land\ Phase2aClassic(c, m, C)$
PROVE: $\land \lor Suggest(m, \langle C \rangle)$
$\lor$ UNCHANGED $\overline{maxTried}$
$\land$ UNCHANGED $\langle propCmd, learned, bA \rangle$

4.1. $\lor Suggest(m, \langle C \rangle)$
$\lor$ UNCHANGED $\overline{maxTried}$

PROOF SKETCH: A *Phase2aClassic* step either increases $\overline{maxTried}[m]$ by C or leaves it as it is. In the first case, it implements a *Suggest* step; in the second it implements a stutering step. We first show conditions that are necessary and sufficient for $\overline{maxTried}$ to stay unchanged on a *Phase2aClassic* step. We then show that if it changes, then a *Suggest* step follows.

4.1.1. ASSUME: $\overline{maxTried}'[m] \neq none$
PROVE: $\overline{maxTried}[m] \neq none$

PROOF: The proof is by contradiction. Suppose that $\overline{maxTried}[m] = none$. Then, by the definitions of *AllTried* and *Tried*, for all coordinator quorum m-coordquorum Q there is a coordinator $d \in Q$ such that $dMaxTried[d][m] = none$, and $Tried(Q, m) = none$. By the assumption, for some m-coordquorum Q, $Tried(Q, m)' \neq none$.

Because $Tried(Q, m) = none$ and $Tried(Q, m)' \neq none$ and only $dMaxTried[c][m]$ was changed, c must be in Q and $dMaxTried[c][m] = none$, but this contradicts a pre-condition of $Phase2aClassic$ steps.

4.1.2. ASSUME: $\overline{maxTried}'[m] \neq \overline{maxTried}[m]$
PROVE: $\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet C$

PROOF: By the assumption, $\overline{maxTried}'[m] \neq none$, and by the step 4.1.1, $\overline{maxTried}[m] \neq none$.

Let QD be the set of m-coordquorums Q such that $Tried(Q, m) \neq Tried(Q, m)'$; clearly, $\forall Q \in QD, c \in Q$. By the assumption, QD is not empty and for all $Q \in QD$, $Tried(Q, m) \sqsubseteq dMaxTried[c][m]$ and $Tried(Q, m)' \sqsubseteq dMaxTried[c][m] \bullet C$. Because only $dMaxTried[c][m]$ was changed and $Tried(Q, m)' \neq Tried(Q, m)$, and by the definition of $\sqcap$, $Tried(Q, m)' = Tried(Q, m) \bullet C$.

Therefore, $\overline{maxTried}' = \sqcup(AllTried(m) \cup \{ Tried(Q, m) \bullet C : Q \in QD\})$, and the set $AllTried(m) \cup \{ Tried(Q, m) \bullet C : Q \in QD\}$ is compatible. Because the first set contains all commands of the second but C, the least upper bound of the first, $\overline{maxTried}$ differs from the least upper bound of the union, $\overline{maxTried}'$ only by the adition of C to the first, and, because they are compatible, $\overline{maxTried}' = \overline{maxTried} \bullet C$

4.1.3. ASSUME: $\overline{maxTried}' = \overline{maxTried} \bullet C$
PROVE: $Suggest(m, \langle C \rangle)$

4.1.3.1. $\langle C \rangle \in Seq(propCmd)$
PROOF: Because $C \in propCmd$ and by the definition of Seq.

4.1.3.2. $\overline{maxTried}[m] \neq none$
PROOF: By step 4.1.1.

4.1.3.3. $\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \langle C \rangle]$
PROOF: By step 4.1.2.

4.1.3.4. Q.E.D.

4.1.4. Q.E.D.

4.2. UNCHANGED $\langle propCmd, bA, learned \rangle$
PROOF: By the definition of $Phase2aClassic$, variables $propCmd$, bA, and $learned$ are kept unchanged.

4.3. Q.E.D.

5. ASSUME: $\wedge\ a \in Acceptor$
$\wedge\ m \in BalNum$
$\wedge\ Phase1b(a, m)$
PROVE: $JoinBallot(a, m)$

PROOF: Any *Phase1b* step clearly implements a *JoinBallot* step, as all the latter's pre and post conditions are also required by the first.

6. ASSUME: $\land\ a \in Acceptor$
$\land\ m \in BalNum$
$\land\ v \in CStruct$
$\land\ Phase2bClassic(a, m, v)$
PROVE: $ClassicVote(a, m, v)$

6.1. $m \geq \widehat{bA}_a$

PROOF: By the definition of *Phase2bClassic*.

6.2. v is safe at round m in bA

PROOF: *Phase2bClassic* has as pre-condition that a has received a "2a" message from all coordinators in some coord-quorum L for round $\widehat{bA}_a$, and that there is a c-struct u such that for all such messages, u is a prefix of the value w in each one. Messages "2a" are sent in actions *Phase2Start* and *Phase2aClassic*, and in both cases the value sent is set to $dMaxTried[c][m]$, where c is the sender coordinator and m is the round in which the message was sent. Because $dMaxTried[c][m] = none$ is a pre-condition to action *Phase2Start* and it changes $dMaxTried[c][m]$ to something different from *none* and no other step changes it back to *none*, a *Phase2Start* step happens only once for a given c and m. A *Phase2aClassic* action is only enabled after this *Phase2Start* step, and it just appends some c-seq to the previous value of $dMaxTried[c][m]$. Therefore, the value set to $dMaxTried[c][m]$ by the *Phase2Start* step is always a prefix of $dMaxTried[c][m]$ in future states. Let $firstTried[c][m]$ be such initial value. By the definition of *Phase2Start* and Proposition 7, $firstTried[c][m]$ is safe at m in bA. Because $firstTried[c][m]$ is a prefix of any value sent by c on "2a" messages in round m, it is also a prefix of u, and u must be safe. Since v is equal to or an extension of u, it is also safe.

6.3. $v \sqsubseteq \overline{maxTried}[m]$

PROOF: Let L be the m-coordquorum from which a has received the "2a" messages in the *Phase2BClassic* step and u be the common lower bound to all values received in such messages according to the action pre-condition. By the definition of *Tried* in $\overline{maxTried}$, by the definition of glb, and because $dMaxTried[c][m]$ is only extended for any coordinator c and balnum m, $u \sqsubseteq Tried(L, m)$. By the definition of *AllTried* in $\overline{maxTried}$ and lub, $Tried(l, m) \sqsubseteq \overline{maxTried}[m]$, and therefore $u \sqsubseteq \overline{maxTried}[m]$. Since $v \sqsubseteq u$, it follows that $v \sqsubseteq \overline{maxTried}[m]$

6.4. $\lor\ bA_a[m] = none$
$\lor\ bA_a[m] \sqsubseteq v$.
PROOF: By the definition of *Phase2bClassic*.
6.5. $bA_a[m]' = v$
PROOF: By the definition of *Phase2bClassic*.
6.6. UNCHANGED $\langle propCmd, \overline{maxTried}, learned\rangle$
PROOF: By Proposition 9 and the definition of *Phase2bClassic*.
6.7. Q.E.D.

7. ASSUME: $\land\ a \in Acceptor$
$\land\ C \in Cmd$
$\land\ Phase2bFast(a, C)$
PROVE: $FastVote(a, C)$
PROOF: Due to Proposition 9, and the definition of *Phase2bFast*, any *Phase2bFast* step is also a *FastVote* step, as all the latter's pre and post conditions are also satisfied by the first.

8. ASSUME: $\land\ l \in Learner$
$\land\ v \in CStruct$
$\land\ Learn(l, v)$
PROVE: $\overline{Learn}(l, v)$
8.1. v is chosen in bA
PROOF: The first pre-condition of *Learn* implies that all acceptors in some m-quorum Q executed action *Phase2bFast* or *Phase2bClassic*. Because commands and c-seqs can only be appended to $bA_a[m]$ in these actions, for all acceptors $a \in Q$, $v \sqsubseteq bA_a[m]$. Therefore, by the definition of **chosen at**, v is chosen at m in bA, and so is v is chosen at bA.
8.2. $\land\ learned'[l] = learned[l] \sqcup v]$
$\land$ UNCHANGED $\langle propCmd, \overline{maxTried}, bA\rangle$
PROOF: By the definition of *Learn*.
8.3. Q.E.D.
PROOF: By the definition of *AbstractLearn* and steps 8.1 and 8.2.
9. Q.E.D.
□

A.4 Multicoordinated Paxos

To prove correctness of the algorithm presented in Section 3.2, we first add the following history variables to the algorithm presented in the previous

section.

crnd An array of balnums, where $crnd[c]$ represents the current round of coordinator c. Initially 0.

cval An array of c-structs, where $cval[c]$ represents the latest c-struct coordinator c has sent in a phase "`2a`" message for round $crnd[c]$. Initially $\bot$.

rnd An array of balnums, where $rnd[a]$ is the current round of acceptor a, that is, the highest-numbered round a has heard of. Initially 0.

vrnd An array of balnums, where $vrnd[a]$ is the round at which acceptor a has accepted the latest value. Initially 0.

vval An array of c-structs, where $vval[a]$ is the c-struct acceptor a has accepted at $vrnd[a]$. Initially $\bot$.

msgs2 A set of messages sent by coordinators and acceptors. This variable is different from the original *msgs* variable.

We now make some simple changes to the algorithm's actions in order to update these history variables accordingly. Notice that the following algorithm is not exactly the same as in the previous section. The preconditions of actions *Phase1a*, *Phase2Start*, and *Phase2aClassic* are slightly more restrictive. However, it is an obvious implementation of the previous version.

Propose(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to $propCmd \cup \{C\}$, from where coordinators and acceptors can read C.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff $\forall j \geq m : dMaxTried[c][j] = none$. It adds message $\langle$"`1a`", $m\rangle$ to *msgs* and *msgs2*.

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

- $\widehat{bA}_a < m$
- $\langle$"`1a`", $m\rangle \in msgs$

It sets $\widehat{bA}_a$ and $rnd[a]$ to m, adds message $\langle$"`1b`", $m, bA_a\rangle$ to *msgs* and message $\langle$"`1b`", $m, vval[a], vrnd[a]\rangle$ to *msgs2*.

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff

- $\forall j \geq m : dMaxTried[c][j] = none$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", $m, bA_a\rangle \in msgs$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, m, \beta)$, and β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = m$ and c has received a message $\langle$"1b", $m, \rho\rangle$ from a with $\rho = \beta_a$.

This action sets $dMaxTried[c][m]$ and $cval[c]$ to v, $crnd[c]$ to m, and adds message $\langle$"2a", $m, v\rangle$ to $msgs$ and $msgs2$.

Phase2aClassic(c, m, C) executed by coordinator c, for command C. The action is enabled iff

- $C \in propCmd$.
- $dMaxTried[c][m] \neq none$
- $\forall j > m : maxTried[c][j] = none$

This action adds message $\langle$"2a", $m, c, dMaxTried[c][m] \bullet C\rangle$ to $msgs$ and $msgs2$, and sets $dMaxTried[c][m]$ and $cval[c]$ to $dMaxTried[c][m]\bullet C$.

Phase2bClassic(a, m, v) executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq \widehat{bA}_a$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $bA_a[m]$ equals $none$ and v equals u, or $bA_a[m]$ and u are compatible and v equals $bA_a[m] \sqcup u$.

It sets $bA_a[m]$ and $vval[a]$ to v, $\widehat{bA}_a$, $rnd[a]$, and $vrnd[a]$ to m, and adds message $\langle$"2b", $m, v\rangle$ to $msgs$ and $msgs2$.

Phase2bFast(a, C) executed by acceptor a, for balnum m and command C. The action is enabled iff

- $\widehat{bA}_a$ is a fast balnum,
- $bA_a[\widehat{bA}_a] \neq none$, and
- $C \in propCmd$.

It sets $bA_a[\widehat{bA}_a]$ and $vval[a]$ to $bA_a[\widehat{bA}_a] \bullet C$ and adds message $\langle \text{“2b”}, \widehat{bA}_a, bA_a[\widehat{bA}_a] \bullet C \rangle$ to $msgs$ and $msgs2$.

Learn(l, v) executed by learner l, for c-struct v. Executed by learner l. It is enabled iff a has received phase “2b” messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

Variables $crnd$, $cval$, rnd, $vrnd$, $vval$, and $msgs2$ appear in no pre-condition and, therefore, are clearly history variables satisfying conditions H1-5 of [1]. This implies that the resulting algorithm is equivalent to (i.e., accepts the same behaviors as) the previous one without such variables. The following invariants can be easily proved for this new algorithm:

InvDA1: $crnd[c] = k \iff \begin{array}{ll} \wedge & dMaxTried[c][k] \neq none \\ \wedge & \forall j > k : dMaxTried[c][j] = none \end{array}$

InvDA2: $cval[c] = dMaxTried[c][crnd[c]]$

InvDA3: $rnd[a] = \widehat{bA}_a$

InvDA4: $vrnd[a] = k \iff \begin{array}{ll} \wedge & bA_a[k] \neq none \\ \wedge & \forall j > k : bA_a[j] = none \end{array}$

InvDA5: $vval[a] = bA_a[vrnd[a]]$

InvDA6: $\langle \text{“1a”}, m \rangle \in msgs \iff \langle \text{“1a”}, m \rangle \in msgs2$

InvDA7: $\langle \text{“1b”}, m, \rho \rangle \in msgs \iff \langle \text{“1b”}, m, vval, vrnd \rangle \in msgs2$, where $vrnd$ is the highest balnum k such that $\rho[k] \neq none$ and $vval$ equals $\rho[vrnd]$.

InvDA8: $\langle \text{“2a”}, m, v \rangle \in msgs \iff \langle \text{“2a”}, m, v \rangle \in msgs2$

InvDA9: $\langle \text{“2b”}, m, v \rangle \in msgs \iff \langle \text{“2b”}, m, v \rangle \in msgs2$

We can use these invariants to rewrite the pre-conditions of the previous algorithm's actions in the following way:

Propose(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to $propCmd \cup \{C\}$, from where coordinators and acceptors can read C.

This action remains the same.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff $crnd[c] < m$. It adds message $\langle$"1a", $m\rangle$ to *msgs* and *msgs2*.

By invariant InvDA1.

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

- $rnd[a] < m$
- $\langle$"1a", $m\rangle \in msgs2$

It sets $\widehat{bA}_a$ and $rnd[a]$ to m, adds message $\langle$"1b", $m, bA_a\rangle$ to *msgs* and message $\langle$"1b", $m, vval[a], vrnd[a]\rangle$ to *msgs2*.

By invariants InvDA3 and InvDA6.

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff

- $crnd[c] < m$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", $m, vval, vrnd\rangle \in msgs2$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, 1bMsg)$ (see *ProvedSafe* as defined in Section 3.2) , and *1Msg* is a mapping from every acceptor a in Q to the phase "1b" message of the previous condition coming from a.

This action sets $dMaxTried[c][m]$ and $cval[c]$ to v, $crnd[c]$ to m, and adds message $\langle$"2a", $m, v\rangle$ to *msgs* and *msgs2*.

By invariants InvDA1 and InvDA6. The equivalence between $ProvedSafe(Q, m, \beta)$ (of Section A.3) and $ProvedSafe(Q, 1bMsg)$ (of Section 3.2) is given by InvDA6.

Phase2aClassic(c, m, C) executed by coordinator c, for command C. The action is enabled iff

- $C \in propCmd$.
- $crnd[c] = m$

This action adds message $\langle$"2a", $m, c, cval[c] \bullet C\rangle$ to *msgs* and *msgs2*, and sets $dMaxTried[c][m]$ and $cval[c]$ to $cval[c] \bullet C$.

By invariants InvDA1 and InvDA2.

Phase2bClassic(a, m, v) executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq rnd[a]$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message (through *msgs2*) for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $vrnd[a] < m$ and v equals u, or $vrnd[a] = m$, $vval[a]$ and u are compatible, and v equals $vval[a] \sqcup u$.

It sets $bA_a[m]$ and $vval[a]$ to v, $\widehat{bA}_a$, $rnd[a]$, and $vrnd[a]$ to m, and adds message $\langle$"2b", $m, v\rangle$ to *msgs* and *msgs2*.

By invariants InvDA3, InvDA4, InvDA8.

Phase2bFast(a, C) executed by acceptor a, for balnum m and command C. The action is enabled iff

- $rnd[a]$ is a fast balnum,
- $rnd[a] = vrnd[a]$, and
- $C \in propCmd$.

It sets $bA_a[\widehat{bA}_a]$ and $vval[a]$ to $vval[a] \bullet C$ and adds message $\langle$"2b", $m, vval[a] \bullet C\rangle$ to *msgs* and *msgs2*.

By invariants InvDA3, InvDA4, InvDA5, and the fact that $rnd[a] \geq vrnd[a]$, which can be inferred by the definition of a ballot array and invariants InvDA3 and InvDA4.

Learn(l, v) executed by learner l, for c-struct v. Executed by learner l. It is enabled iff a has received (through *msgs2*) phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

By invariant InvDA9.

The resulting algorithm now has variables bA, *dMaxTried* and *msgs* as history variables, since they do not appear on any action's pre-condition and are only updated. This algorithm is, therefore, equivalent to one that does not contain such variables, which we present below.

Propose(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to $propCmd \cup \{C\}$, from where coordinators and acceptors can read C.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff $crnd[c] < m$. It adds message $\langle$"`1a`", $m\rangle$ to *msgs2*.

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

- $rnd[a] < m$
- $\langle$"`1a`", $m\rangle \in msgs2$

It sets $rnd[a]$ to m, and adds message $\langle$"`1b`", $m, vval[a], vrnd[a]\rangle$ to *msgs2*.

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff

- $crnd[c] < m$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"`1b`", $m, vval, vrnd\rangle \in msgs2$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, 1bMsg)$, and *1Msg* is a mapping from every acceptor a in Q to the phase "`1b`" message of the previous condition coming from a.

This action sets $cval[c]$ to v, $crnd[c]$ to m, and adds message $\langle$"`2a`", $m, v\rangle$ to *msgs2*.

Phase2aClassic(c, m, C) executed by coordinator c, for command C. The action is enabled iff

- $C \in propCmd$.
- $crnd[c] = m$

This action adds message $\langle$"`2a`", $m, c, cval[c] \bullet C\rangle$ to *msgs2*, and sets $cval[c]$ to $cval[c] \bullet C$.

Phase2bClassic(a, m, v) executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq rnd[a]$,

- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message (through *msgs2*) for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $vrnd[a] < m$ and v equals u, or $vrnd[a] = m$, $vval[a]$ and u are compatible, and v equals $vval[a] \sqcup u$.

It sets $vval[a]$ to v, $rnd[a]$ and $vrnd[a]$ to m, and adds message $\langle$"2b", $m, v\rangle$ to *msgs2*.

Phase2bFast(a, C) executed by acceptor a, for balnum m and command C. The action is enabled iff

- $rnd[a]$ is a fast balnum,
- $rnd[a] = vrnd[a]$, and
- $C \in propCmd$.

It sets $vval[a]$ to $vval[a] \bullet C$ and adds message $\langle$"2b", $m, vval[a] \bullet C\rangle$ to *msgs2*.

Learn(l, v) executed by learner l, for c-struct v. Executed by learner l. It is enabled iff a has received (through *msgs2*) phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

The algorithm presented in Section 3.2 is a stricter implementation of the algorithm above, which can be easily verified by simply comparing their actions. This concludes the proof that Multicoordinated Paxos satisfies the safety requirements of Generalized Consensus. Section B.4 presents the unambiguous TLA$^+$ specification of the basic algorithm presented in Section 3.2 and Section B.5 presents its complete TLA$^+$ specification including actions for collision recovery and a simplified version of the mechanism for reducing the number of disk writes presented in Section 4.4.

A.5 Collision Recovery

The mechanisms for collision recovery described in Section 4.2 simply simulate the execution of basic actions of the algorithm in a way compatible with the actual behavior. The interpretation of a collision as a phase "1a" message for the following balnum, for example, simulates a *Phase1a* action for that balnum. Since action *Phase1a* simply sends a "1a" message with

no guarantee of delivery, the process detecting the collision could be the only one receiving such a message. Given that these basic actions are already proved correct, such mechanisms cannot the safety properties of our specification.

A.6 Liveness

That the basic algorithm provides means for ensuring liveness is easy to see, since new balnums can always be started and a classic balnum has the same liveness requirements as classic Paxos. The discussion in Section 4.3 extends Multicoordinated Paxos in this sense and sketches its liveness conditions. A rigorous liveness proof will appear elsewhere.

B TLA$^+$ Specifications

B.1 Helper Specifications

B.1.1 Order Relations

This module was defined in [9].

MODULE *OrderRelations*

We make some definitions for an arbitrary ordering relation $\sqsubseteq$ on a set S. The module will be used by *instantiang* $\sqsubseteq$ and S with a particular operator and Set.

CONSTANTS S, $\_ \sqsubseteq \_$

We define *IsPartialOrder* to be the assertion that $\sqsubseteq$ is an (irreflexive) partial order on a set S, and *IsTotalOrder* to be the assertion that it is a total ordering of S.

$$\begin{aligned}
&\mathit{IsPartialOrder} \triangleq \\
&\quad \land \forall\, u,\, v,\, w \in S : (u \sqsubseteq v) \land (v \sqsubseteq w) \Rightarrow (u \sqsubseteq w) \\
&\quad \land \forall\, u,\, v \in S \quad : (u \sqsubseteq v) \land (v \sqsubseteq u) \Rightarrow (u = v)
\end{aligned}$$

$$\begin{aligned}
&\mathit{IsTotalOrder} \triangleq \\
&\quad \land \mathit{IsPartialOrder} \\
&\quad \land \forall\, u,\, v \in S : (u \sqsubseteq v) \lor (v \sqsubseteq u)
\end{aligned}$$

We now define the glb (greatest lower bound) and lub (least upper bound) operators. To define *GLB*, we first define $\mathit{IsLB}(lb,\, T)$ to be true iff lb is a lower bound of T, and $\mathit{IsGLB}(lb,\, T)$ to be true iff lb is a glb of T. the value of $\mathit{GLB}(T)$ is unspecified if T has no glb. The definition for upper bounds are analogous.

$$\begin{aligned}
&\mathit{IsLB}(lb,\, T) \triangleq \land\, lb \in S \\
&\qquad\qquad\qquad\quad \land \forall\, v \in T : lb \sqsubseteq v \\
&\mathit{IsGLB}(lb,\, T) \triangleq \land\, \mathit{IsLB}(lb,\, T) \\
&\qquad\qquad\qquad\quad \land \forall\, v \in S : \mathit{IsLB}(v,\, T) \Rightarrow (v \sqsubseteq lb) \\
&\mathit{GLB}(T) \triangleq \text{CHOOSE } lb \in S : \mathit{IsGLB}(lb,\, T) \\
&v \sqcap w \triangleq \mathit{GLB}(\{v,\, w\}) \\
&\mathit{IsUB}(ub,\, T) \triangleq \land\, ub \in S \\
&\qquad\qquad\qquad\quad \land \forall\, v \in T : v \sqsubseteq ub \\
&\mathit{IsLUB}(ub,\, T) \triangleq \land\, \mathit{IsUB}(ub,\, T) \\
&\qquad\qquad\qquad\quad \land \forall\, v \in S : \mathit{IsUB}(v,\, T) \Rightarrow (ub \sqsubseteq v) \\
&\mathit{LUB}(T) \triangleq \text{CHOOSE } ub \in S : \mathit{IsLUB}(ub,\, T) \\
&v \sqcup w \triangleq \mathit{LUB}(\{v,\, w\})
\end{aligned}$$

B.1.2 Command Structs

This module was defined in [9].

MODULE *CStructs*

EXTENDS *Sequences* The *Sequences* module defines the operation *Seq*

We declare the assumed objects as parameters. We use *Bottom* instead of $\bot$.

CONSTANTS Cmd, $CStruct$, $\_ \bullet \_$, $Bottom$

We write $v ** \sigma$ as the overloaded version of $v \bullet \sigma$ for a command sequence σ. The recursive definition below defines the function $conc[w, t] = w ** t$.

$$
\begin{array}{l}
v ** s \triangleq \\
\quad \text{LET } conc[w \in CStruct,\ t \in Seq(Cmd)] \triangleq \\
\qquad \text{IF } t = \langle\rangle \text{ THEN } w \\
\qquad\qquad \text{ELSE } conc[w \bullet Head(t),\ Tail(t)] \\
\quad \text{IN } conc[v,\ s]
\end{array}
$$

$$Str(P) \triangleq \{Bottom ** s : s \in Seq(P)\}$$

Our algorithms use a value *none* that is not a c-struct and extend the relation $\sqsubseteq$ to the element *none* so that $none \sqsubseteq none$, $none \not\sqsubseteq v$, and $v \not\sqsubseteq none$ for any c-struct v.

$$
\begin{array}{l}
none \triangleq \text{CHOOSE } n : n \notin CStruct \\
v \sqsubseteq w \triangleq \ \lor \land v \in CStruct \\
\qquad\qquad\qquad\ \land w \in CStruct \\
\qquad\qquad\qquad\ \land \exists\, s \in Seq(Cmd) : w = v ** s \\
\qquad\qquad\ \lor \land v = none \\
\qquad\qquad\qquad\ \land w = none \\
v \sqsubset w \triangleq (v \sqsubseteq w) \land (v \neq w)
\end{array}
$$

We now import the definitions of the *OrderRelations* module with *CStruct* substituted for S and $\sqsubseteq$ substituted for $\preceq$

INSTANCE *OrderRelations* WITH $S \leftarrow CStruct$

We now define compatibility of c-structs and of sets of c-structs, and of contains, giving them obvious operator nomes.

$$
\begin{array}{ll}
AreCompatible(v,\ w) & \triangleq \exists\, ub \in CStruct : IsUB(ub,\ \{v,\ w\}) \\
IsCompatible(S) & \triangleq \forall\, v,\ w \in S : AreCompatible(v,\ w) \\
Contains(v,\ C) & \triangleq \exists\, s,\ t \in Seq(Cmd) : v = ((Bottom ** s) \bullet C) ** t
\end{array}
$$

Here are the formal statements of assumptions *CS*0-*CS*4.

$$CS0 \triangleq \forall\, v \in CStruct,\ C \in Cmd : v \bullet C \in CStruct$$

$CS1 \triangleq CStruct = Str(Cmd)$

$CS2 \triangleq IsPartialOrder$

$$\begin{array}{l} CS3 \triangleq \forall\, P \in \text{SUBSET } Cmd \setminus \{\{\}\} : \\ \quad \land \forall\, v,\, w \in Str(P) : \\ \quad\quad \land\, v \sqcap w \in Str(P) \\ \quad\quad \land\, IsGLB(v \sqcap w,\, \{v,\, w\}) \\ \quad\quad \land\, AreCompatible(v,\, w) \Rightarrow \land\, v \sqcup w \in Str(P) \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\, IsLUB(v \sqcup w,\, \{v,\, w\}) \end{array}$$

$$\begin{array}{l} CS4 \triangleq \forall\, v,\, w \in CStruct,\, C \in Cmd : \\ \quad AreCompatible(v,\, w) \land Contains(v,\, C) \land Contains(w,\, C) \Rightarrow \\ \quad\quad Contains(v \sqcap w,\, C) \end{array}$$

ASSUME $CS0 \land CS1 \land CS2 \land CS3 \land CS4$

B.1.3 Paxos Constants

This module was defined in [9], but was extended as needed to define multicoordinated algorithms. For example, it was extended with the definition of $CoordQuorum(m)$.

MODULE *PaxosConstants*

This module defines the data structures for the abstract algorithm. It is basically the same module *PaxosConstants* found in the Generalized Paxos paper, except for the introduction of constants *Coord* and $CoordQuorum(\_)$, and assumption *CoordQuorumAssumption*.

EXTENDS *CStructs*, *FiniteSets*

Module *FiniteSets* defines $IsFiniteSet(S)$ to be true iff S is a finite set

We introduce the parameter *IsFast*, where $IsFast(m)$ is true iff m is a fast ballot number. The ordering relation $\preceq$ on ballot numbers is also a parameter.

CONSTANTS *BalNum*, *Zero*, $\_ \preceq \_$, $IsFast(\_)$

We assume that *Zero* is the first balnum, and that $\preceq$ is a total ordering of the set *BalNum* of balnums.

$$\begin{array}{l} \text{ASSUME } \land\, Zero \in BalNum \\ \quad\quad \land \text{ LET } PO \triangleq \text{ INSTANCE } OrderRelations \text{ WITH } S \leftarrow BalNum,\ \sqsubseteq \leftarrow \preceq \\ \quad\quad\quad \text{IN } \quad PO!IsTotalOrder \end{array}$$

We define $i \prec j$ to be true iff $i \preceq j$ for two different balnums i and j

$i \prec j \triangleq (i \preceq j) \land (i \neq j)$

If B is a set of ballot numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq \text{CHOOSE } i \in B : \forall\, j \in B : j \preceq i$

This section of the module is the only part that differs the original *PaxosConstants* module presented in the Generalized *Paxos* paper. We have added the constants *Coord* and *CoordQuorum*, and the assumptions that coordinator quorums are sets of coordinators which intersect if they refer to the same ballot number.

CONSTANTS $Learner$, $Acceptor$, $Quorum(\_)$, $Coord$, $CoordQuorum(\_)$

$$
\begin{array}{l}
QuorumAssumption \triangleq \\
\quad \forall\, i \in BalNum : \\
\qquad \land\ Quorum(i) \subseteq \text{SUBSET } Acceptor \\
\qquad \land\ \forall\, j \in BalNum : \\
\qquad\quad \land\ \forall\, Q \in Quorum(i),\ R \in Quorum(j) : Q \cap R \neq \{\} \\
\qquad\quad \land\ IsFast(j) \Rightarrow \\
\qquad\qquad \forall\, Q \in Quorum(i),\ R1,\ R2 \in Quorum(j) : \\
\qquad\qquad\quad Q \cap R1 \cap R2 \neq \{\}
\end{array}
$$

ASSUME $QuorumAssumption$

$$
\begin{array}{l}
CoordQuorumAssumption \triangleq \\
\quad \forall\, i \in BalNum : \\
\qquad \land\ CoordQuorum(i) \subseteq \text{SUBSET } Coord \\
\qquad \land\ \forall\, Q,\ R \in CoordQuorum(i) : Q \cap R \neq \{\}
\end{array}
$$

ASSUME $CoordQuorumAssumption$

We define *BallotArray* to be the set of all ballot arrays. We represent a ballot array as a record, where we write $\beta_a[m]$ as $\beta.vote[m]$ and $\hat{\beta}_a$ as $\beta.mbal[a]$.

$$
\begin{array}{l}
BallotArray \triangleq \\
\quad \{beta \in [vote : [Acceptor \to [BalNum \to CStruct \cup \{none\}]], \\
\qquad\qquad mbal : [Acceptor \to BalNum]] : \\
\qquad \forall\, a \in Acceptor : \\
\qquad\quad \land\ beta.vote[a][Zero] \neq none \\
\qquad\quad \land\ IsFiniteSet(\{m \in BalNum : beta.vote[a][m] \neq none\}) \\
\qquad\quad \land\ \forall\, m \in BalNum : (beta.mbal[a] \prec m) \Rightarrow (beta.vote[a][m] = none)\}
\end{array}
$$

We now formalize the definitions of *chosen at*, *safe at*, etc. We translate the *English* terms into obvious operator names. For example, $IsChosenAt(v,\ m,\ \beta\ )$ is defined to be true iff v is chosen at m in β , assuming that v is a c-struct, m is a balnum, and β a ballot array. (We don't care what $IsChosenAt(v,\ m,\ \beta\ )$ means for other values of v, m, and β.) We also assert the three propositions as theorems.

$IsChosenAt(v,\ m,\ beta) \triangleq \exists\, Q \in Quorum(m) : \forall\, a \in Q : (v \sqsubseteq beta.vote[a][m])$

$IsChosenIn(v,\ beta) \triangleq \exists\, m \in BalNum : IsChosenAt(v,\ m,\ beta)$

$IsChoosableAt(v,\ m,\ beta) \triangleq$
$\quad \exists\, Q \in Quorum(m) :$
$\quad\quad \forall\, a \in Q : (m \prec beta.mbal[a]) \Rightarrow (v \sqsubseteq beta.vote[a][m])$

$IsSafeAt(v,\ m,\ beta) \triangleq$
$\quad \forall\, k \in BalNum :$
$\quad\quad (k \prec m) \Rightarrow \forall\, w \in CStruct : IsChoosableAt(w,\ k,\ beta) \Rightarrow (w \sqsubseteq v)$

$IsSafe(beta) \triangleq$
$\quad \forall\, a \in Acceptor,\ k \in BalNum :$
$\quad\quad (beta.vote[a][k] \neq none) \Rightarrow IsSafeAt(beta.vote[a][k],\ k,\ beta)$

THEOREM Proposition 1
$\quad \forall\, beta \in BallotArray :$
$\quad\quad IsSafe(beta) \Rightarrow IsCompatible(\{v \in CStruct : IsChosenIn(v,\ beta)\})$

$ProvedSafe(Q,\ m,\ beta) \triangleq$
$\quad$ LET $k \triangleq Max(\{i \in BalNum :$
$\qquad\qquad\qquad (i \prec m) \wedge (\exists\, a \in Q : beta.vote[a][i] \neq none)\})$
$\qquad RS \triangleq \{R \in Quorum(k) : \forall\, a \in Q \cap R : beta.vote[a][k] \neq none\}$
$\qquad g(R) \triangleq GLB(\{beta.vote[a][k] : a \in Q \cap R\})$
$\qquad G \triangleq \{g(R) : R \in RS\}$
$\quad$ IN IF $RS = \{\}$ THEN $\{beta.vote[a][k] :$
$\qquad\qquad\qquad\qquad a \in \{b \in Q : beta.vote[b][k] \neq none\}\}$
$\qquad\qquad$ ELSE IF $IsCompatible(G)$ THEN $\{LUB(G)\}$
$\qquad\qquad\qquad\qquad\qquad\qquad$ ELSE $\{\}$

THEOREM Proposition 2
$\quad \forall\, m \in BalNum \setminus \{Zero\},\ beta \in BallotArray :$
$\quad\quad \forall\, Q \in Quorum(m) :$
$\quad\quad\quad \wedge\ IsSafe(beta)$
$\quad\quad\quad \wedge \forall\, a \in Q : m \preceq beta.mbal[a]$
$\quad\quad\quad \Rightarrow \forall\, v \in ProvedSafe(Q,\ m,\ beta) : IsSafeAt(v,\ m,\ beta)$

$IsConservative(beta) \triangleq$
$\quad \forall\, m \in BalNum,\ a,\ b \in Acceptor :$
$\quad\quad \wedge \neg IsFast(m)$
$\quad\quad \wedge\ beta.vote[a][m] \neq none$
$\quad\quad \wedge\ beta.vote[b][m] \neq none$

$\Rightarrow AreCompatible(beta.vote[a][m],\ beta.vote[b][m])$

THEOREM Proposition 3

$\forall\, beta \in BallotArray :$
$\quad IsConservative(beta) \Rightarrow$
$\quad\quad \forall\, m \in BalNum \setminus \{Zero\} :$
$\quad\quad\quad \forall\, Q \in Quorum(m) : ProvedSafe(Q,\ m,\ beta) \neq \{\}$

B.2 Abstract Multicoordinated Paxos

This module specifies an abstract version of the Multicoordinated Paxos algorithm. It is used as the first step in the proof of correctness of Multicoordinated Paxos.

MODULE *AbstractMCPaxos*

This module specifies the Abstract MultiCoordinated Paxos algorithm. It resembles the specification of the Abstract Generalized Paxos algorithm presented in the Generalized Paxos paper, but some changes are required since we do not have a *minTried* variable.

EXTENDS *PaxosConstants*

VARIABLES *propCmd*, *maxTried*, *bA*, *learned*

Invariants

Type invariant.

$$\begin{aligned} TypeInv \triangleq\ & \land propCmd \subseteq Cmd \\ & \land learned \in [Learner \to CStruct] \\ & \land bA \in BallotArray \\ & \land maxTried \in [BalNum \to CStruct \cup \{none\}] \end{aligned}$$

Other invariants satisfied by the algorithm.

$maxTriedInvariant \triangleq$
$\quad \forall\, m \in BalNum :$
$\quad\quad (maxTried[m] \neq none) \Rightarrow$
$\quad\quad\quad \land maxTried[m] \in Str(propCmd)$
$\quad\quad\quad \land IsSafeAt(maxTried[m],\ m,\ bA)$

$bAInvariant \triangleq$
$\quad \forall\, a \in Acceptor,\ m \in BalNum :$
$\quad\quad (bA.vote[a][m] \neq none) \Rightarrow$
$\quad\quad\quad \land IsSafeAt(bA.vote[a][m],\ m,\ bA)$
$\quad\quad\quad \land \neg IsFast(m) \Rightarrow (bA.vote[a][m] \sqsubseteq maxTried[m])$

$$\land\ IsFast(m) \Rightarrow (bA.vote[a][m] \in Str(propCmd))$$

$learnedInvariant \triangleq$

$\forall\, l \in Learner :\ \land\ learned[l] \in Str(propCmd)$

$\land\ \exists\, S \in \text{SUBSET}\ CStruct :$

$\land\ IsFiniteSet(S)$

$\land\ \forall\, v \in S : IsChosenIn(v,\ bA)$

$\land\ learned[l] = LUB(S)$

Actions

$Propose(C)$ specifies the action of proposing command C

$Propose(C) \triangleq$

$\land\ \ C \notin propCmd$

$\land\ \ propCmd' = propCmd \cup \{C\}$

$\land\ \ \text{UNCHANGED}\ \langle maxTried,\ bA,\ learned\rangle$

Action $JoinBallot(a,\ m)$ increases the current ballot number of agent a, setting it to m.

$JoinBallot(a,\ m) \triangleq$

$\land\ bA.mbal[a] \prec m$

$\land\ bA' = [bA\ \text{EXCEPT}\ !.mbal[a] = m]$

$\land\ \text{UNCHANGED}\ \langle propCmd,\ maxTried,\ learned\rangle$

Action $StartBallot(m, w)$ changes $maxTried[m]$ from $none$ to w, where w is proposed and safe at m in bA.

$StartBallot(m,\ w) \triangleq$

$\land\ maxTried[m] = none$

$\land\ IsSafeAt(w,\ m,\ bA)$

$\land\ w \in Str(propCmd)$

$\land\ maxTried' = [maxTried\ \text{EXCEPT}\ ![m] = w]$

$\land\ \text{UNCHANGED}\ \langle propCmd,\ bA,\ learned\rangle$

Action $Suggest(m, \sigma)$ extends $maxTried[m]$ with σ if $maxTried[m] \neq none$.
The expression $[maxTried\ \text{EXCEPT}\ ![m] = @\ **s]$ represents a vector (in fact, it is a function) which is almost the same as $maxTried$ except for entry m ($![m]$ in the expression), which is set to the previous value of that entry ($@$ in the expression) extended with command sequence s.

$Suggest(m,\ s) \triangleq$

$\land\ \ s \in Seq(propCmd)$

$\land\ \ maxTried[m] \neq none$

$\land\ \ maxTried' = [maxTried\ \text{EXCEPT}\ ![m] = @ ** s]$

$\land\ \ \text{UNCHANGED}\ \langle propCmd,\ bA,\ learned\rangle$

Action $ClassicVote(a,\ v)$ sets $\widehat{bA}_a$ to m and $bA_a[m]$ to v if
(i) $\widehat{bA}_a \preceq m$
(ii) v is safe at m in bA,
(iii) $v \sqsubseteq maxTried[m]$, and
(iv) either $bA_a[m]$ equals $none$ or v is an extension of it.

$$
\begin{aligned}
&ClassicVote(a,\ m,\ v) \triangleq \\
&\quad \land\ bA.mbal[a] \preceq m \\
&\quad \land\ IsSafeAt(v,\ m,\ bA) \\
&\quad \land\ v \sqsubseteq maxTried[m] \\
&\quad \land\ \lor\ bA.vote[a][m] = none \\
&\qquad \lor\ bA.vote[a][m] \sqsubseteq v \\
&\quad \land\ bA' = [bA \text{ EXCEPT } !.mbal[a] = m,\ !.vote[a][m] = v] \\
&\quad \land \text{ UNCHANGED } \langle propCmd,\ maxTried,\ learned \rangle
\end{aligned}
$$

Action $FastVote(a,\ C)$ extends $bA_a[\widehat{bA}_a]$ with proposed command C if $\widehat{bA}_a$ is a fast balnum and $bA_a[\widehat{bA}_a] \neq none$. Expression $[bA \text{ EXCEPT } !.vote[a][bA.mbal[a]] = @ \bullet C]$ follows the same principle explained in action $Suggest(m,\ C)$.

$$
\begin{aligned}
&FastVote(a,\ C) \triangleq \\
&\quad \land\ C \in propCmd \\
&\quad \land\ IsFast(bA.mbal[a]) \\
&\quad \land\ bA.vote[a][bA.mbal[a]] \neq none \\
&\quad \land\ bA' = [bA \text{ EXCEPT } !.vote[a][bA.mbal[a]] = @ \bullet C] \\
&\quad \land \text{ UNCHANGED } \langle propCmd,\ maxTried,\ learned \rangle
\end{aligned}
$$

Action $AbstractLearn(l,\ v)$ extends $learned[l]$ to the least upper bound of $learned[l]$ and v, if v is chosen.

$$
\begin{aligned}
&AbstractLearn(l,\ v) \triangleq \\
&\quad \land\ IsChosenIn(v,\ bA) \\
&\quad \land\ learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v] \\
&\quad \land \text{ UNCHANGED } \langle propCmd,\ maxTried,\ bA \rangle
\end{aligned}
$$

Complete Specification

Initial predicate

$$
\begin{aligned}
Init \triangleq\ & \land\ propCmd = \{\} \\
& \land\ learned\ \ = [l \in Learner \mapsto Bottom] \\
& \land\ bA = [vote \mapsto \\
& \qquad [a \in Acceptor \mapsto \\
& \qquad\ [m \in BalNum \mapsto \text{IF } m = Zero \text{ THEN } Bottom \\
& \qquad\qquad \text{ELSE } none]], \\
& \quad mbal \mapsto [a \in Acceptor \mapsto Zero]]
\end{aligned}
$$

$\land\ maxTried = [m \in BalNum \mapsto$
IF $m = Zero$ THEN $Bottom$ ELSE $none]$

Actions combined into the next-state relation.

$Next \triangleq \lor \exists\, C \in Cmd : Propose(C)$
$\lor \exists\, a \in Acceptor,\ m \in BalNum : JoinBallot(a,\ m)$
$\lor \exists\, m \in BalNum,\ w \in CStruct : StartBallot(m,\ w)$
$\lor \exists\, m \in BalNum,\ s \in Seq(Cmd) : Suggest(m,\ s)$
$\lor \exists\, a \in Acceptor,\ C \in Cmd : FastVote(a,\ C)$
$\lor \exists\, a \in Acceptor,\ m \in BalNum,\ v \in CStruct :$
$ClassicVote(a,\ m,\ v)$
$\lor \exists\, l \in Learner,\ v \in CStruct : AbstractLearn(l,\ v)$

We define *Spec* to be the complete specification.

$Spec \triangleq Init \land \Box[Next]_{\langle propCmd,\ learned,\ bA,\ maxTried\rangle}$

The following theorem asserts the invariance of our invariants

THEOREM
$Spec \Rightarrow \Box(TypeInv \land maxTriedInvariant \land bAInvariant \land learnedInvariant)$

The following asserts that our specification *Spec* implies/implements the specification Spec from module *GeneralConsensus*.

$GC \triangleq$ INSTANCE $GeneralConsensus$
THEOREM $Spec \Rightarrow GC!Spec$

B.3 Distributed Abstract Multicoordinated Paxos

This module specifies a distributed version of the abstract algorithm in the previous section. It adds the exchange of messages and shows how the distributed data structures can be mapped back to the non-distributed algorithm. It is used as the second step in the proof of correctness of Multicoordinated Paxos.

MODULE *DistAbsMCPaxos*

EXTENDS *PaxosConstants*

The following variables are similar to those in *AbstractMCPaxos* module. They are changed in this module as they are in *AbstractMCPaxos*.

VARIABLES $propCmd,\ bA,\ learned$

We describe the state of the message-passing system by the value of the variable *msgs*. *I.e.*, processes send messages to each othe by simply putting them in the *msgs* variable.

As we do not specify liveness for this protocol, we do not explicitly model message loss. Because no action is required to happen, messages can simply be ignored, modeling the loss of a message by never executing the action that would read the message.

Message duplication is modeled by not removing the message from *msgs* once it is read.

VARIABLE $msgs$

We define *Msg* to be the set of all possible messages. For the sake of clarity and avoiding errors, we let messages be records instead of tuples. For example, the message $\langle$ "2a", m, $v\rangle$ in the text becomes a record with type field "2a", *bal* field m, and *val* field v.

$$\begin{array}{lll} Msg \triangleq & & [type : \{\text{"1a"}\},\ bal : BalNum] \\ & \cup & [type : \{\text{"1b"}\},\ bal : BalNum,\ acc : Acceptor, \\ & & \quad vote : [BalNum \to CStruct \cup \{none\}]] \\ & \cup & [type : \{\text{"2a"}\},\ bal : BalNum,\ val : CStruct,\ coord : Coord] \\ & \cup & [type : \{\text{"2b"}\},\ bal : BalNum,\ acc : Acceptor,\ val : CStruct] \end{array}$$

dMaxTried is a distributed version of *maxTried*. Each coordinator stores in *dMaxTried* the longest *c*-struct it has send in a round, and *MaxTried*, below, will map it to *maxTried*.

VARIABLE $dMaxTried$

Refinement Mapping

dMaxTried is mapped to *maxTried* by the *MaxTried* operator. It maps the *c*-structs tried by all coordinators in some *coord*-quorum $$Q$$ in some round $$m$$ to a single *c*-struct $Tried(Q, m)$. The set of all $Tried(Q, m)$, $AllTried(m)$, is then mapped to *maxTried*.

$$\begin{array}{l} MaxTried \triangleq \\ \quad \text{LET } Tried(Q, m) \triangleq \text{IF } \exists\, c \in Q : dMaxTried[c][m] = none \\ \qquad\qquad\qquad\qquad \text{THEN } none \\ \qquad\qquad\qquad\qquad \text{ELSE } GLB(\{dMaxTried[c][m] : c \in Q\}) \\ \qquad AllTried(m) \triangleq \{v \in \{Tried(Q, m) : Q \in CoordQuorum(m)\} \\ \qquad\qquad\qquad\qquad : v \neq none\} \\ \quad \text{IN } [m \in BalNum \mapsto \text{IF } AllTried(m) = \{\} \\ \qquad\qquad\qquad\qquad \text{THEN } none \\ \qquad\qquad\qquad\qquad \text{ELSE } LUB(AllTried(m))] \end{array}$$

Invariants

Type invariant. implies Abstract!*TypeInv*.

$$
\begin{aligned}
\mathit{TypeInv} \triangleq\ & \land\ \mathit{propCmd} \subseteq \mathit{Cmd} \\
& \land\ \mathit{learned} \in [\mathit{Learner} \to \mathit{CStruct}] \\
& \land\ \mathit{bA} \in \mathit{BallotArray} \\
& \land\ \mathit{dMaxTried} \in [\mathit{Coord} \to [\mathit{BalNum} \to \mathit{CStruct} \cup \{\mathit{none}\}]] \\
& \land\ \mathit{msgs} \subseteq \mathit{Msg}
\end{aligned}
$$

Actions

Action $\mathit{Propose}(C)$ adds command C to $\mathit{propCmd}$.

It implements $\mathit{AbstractMCPaxos}!\mathit{Propose}(C)$ directly.

$$
\begin{aligned}
&\mathit{Propose}(C) \triangleq \\
&\quad \land\ C \notin \mathit{propCmd} \\
&\quad \land\ \mathit{propCmd}' = \mathit{propCmd} \cup \{C\} \\
&\quad \land\ \text{UNCHANGED } \langle \mathit{bA},\ \mathit{learned},\ \mathit{msgs},\ \mathit{dMaxTried} \rangle
\end{aligned}
$$

Action $\mathit{Phase1a}(c,\ m)$ executes phase $1a$ for round m at coordinator c, sending a "1a" message to all acceptors.

It has no counterpart on $\mathit{AbstractMCPaxos}$.

$$
\begin{aligned}
&\mathit{Phase1a}(c,\ m) \triangleq \\
&\quad \land\ \mathit{dMaxTried}[c][m] = \mathit{none} \\
&\quad \land\ \mathit{msgs}' = \mathit{msgs} \cup \{[\mathit{type} \mapsto \text{“1a”},\ \mathit{bal} \mapsto m]\} \\
&\quad \land\ \text{UNCHANGED } \langle \mathit{propCmd},\ \mathit{bA},\ \mathit{learned},\ \mathit{dMaxTried} \rangle
\end{aligned}
$$

Action $\mathit{Phase1b}(a,\ m)$ executes phase $1b$ for round m at acceptor a.

It implements $\mathit{JoinBallot}(a,\ m)$, and sends a "1b" message to coordinators.

$$
\begin{aligned}
&\mathit{Phase1b}(a,\ m) \triangleq \\
&\quad \land\ \mathit{bA}.\mathit{mbal}[a] \prec m \\
&\quad \land\ [\mathit{type} \mapsto \text{“1a”},\ \mathit{bal} \mapsto m] \in \mathit{msgs} \\
&\quad \land\ \mathit{bA}' = [\mathit{bA} \text{ EXCEPT } !.\mathit{mbal}[a] = m] \\
&\quad \land\ \mathit{msgs}' = \mathit{msgs} \cup \\
&\qquad\qquad \{[\mathit{type} \mapsto \text{“1b”},\ \mathit{bal} \mapsto m,\ \mathit{acc} \mapsto a,\ \mathit{vote} \mapsto \mathit{bA}.\mathit{vote}[a]]\} \\
&\quad \land\ \text{UNCHANGED } \langle \mathit{propCmd},\ \mathit{learned},\ \mathit{dMaxTried} \rangle
\end{aligned}
$$

Action $\mathit{Phase2Start}(c,\ m,\ v)$ starts phase $2a$ for round m at coordinator c, proposing the c-struct v.

It implements $\mathit{AbstractMCPaxos}!\mathit{StartBallot}(m,\ \mathit{MaxTried}'[m])$. Because not all $\mathit{Phase2Start}$ change $\mathit{MaxTried}[m]$, some steps are stuttering regarding $\mathit{AbstractMCPaxos}$.

$$
\begin{aligned}
&\mathit{Phase2Start}(c,\ m,\ v) \triangleq \\
&\quad \land\ \mathit{dMaxTried}[c][m] = \mathit{none}
\end{aligned}
$$

$$
\begin{array}{l}
\land\, \exists\, Q \in Quorum(m) : \\
\quad \land\, \forall\, a \in Q : \exists\, msg \in msgs : \land\, msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal\ = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc\ = a \\
\quad \land\, \text{LET}\ 1bMsg \triangleq [a \in Q \mapsto \text{CHOOSE}\ msg \in msgs : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal\ = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc\ = a] \\
\qquad\qquad beta \triangleq [vote \mapsto [a \in Q \mapsto 1bMsg[a].vote], \\
\qquad\qquad\qquad\quad mbal \mapsto [a \in Q \mapsto m]] \\
\qquad \text{IN}\quad \exists\, w \in ProvedSafe(Q,\, m,\, beta),\, s \in Seq(propCmd) : \\
\qquad\qquad \land\, v = w \mathbin{**} s \\
\qquad\qquad \land\, dMaxTried' = [dMaxTried\ \text{EXCEPT}\ ![c][m] = v] \\
\qquad\qquad \land\, msgs' = msgs \cup \\
\qquad\qquad\qquad\qquad\qquad \{[type \mapsto \text{“2a”},\, bal \mapsto m,\, val \mapsto v,\, coord \mapsto c]\} \\
\land\, \text{UNCHANGED}\ \langle propCmd,\, bA,\, learned\rangle
\end{array}
$$

Action $Phase2aClassic(c,\, m,\, C)$ is executed by coordinator c of ballot m, for command C. It adds the command C to c's previously tried value and forward the new value to the acceptors.

It implements the $AbstractMCPaxos!Suggest(m,\, \langle C\rangle)$ action.

$$
\begin{array}{l}
Phase2aClassic(c,\, m,\, C) \triangleq \\
\quad \land\, C \in propCmd \\
\quad \land\, dMaxTried[c][m] \neq none \\
\quad \land\, dMaxTried' = [dMaxTried\ \text{EXCEPT}\ ![c][m] = dMaxTried[c][m] \mathbin{**} C] \\
\quad \land\, msgs' = msgs \cup \\
\qquad\quad \{[type \mapsto \text{“2a”},\, bal \mapsto m,\, val \mapsto dMaxTried'[c][m],\, coord \mapsto c]\} \\
\quad \land\, \text{UNCHANGED}\ \langle propCmd,\, bA,\, learned\rangle
\end{array}
$$

Action $Phase2bClassic(a,\, m,\, v)$ is executed by acceptor a in round m to accept v.

It implements action $AbstractMCPaxos!ClassicVote(a,\, m,\, v)$.

$$
\begin{array}{l}
Phase2bClassic(a,\, m,\, v) \triangleq \\
\quad \land\, bA.mbal[a] \preceq m \\
\quad \land\, \exists\, L \in CoordQuorum(m),\, u \in CStruct : \\
\qquad \land\, \forall\, c \in L : \exists\, msg \in msgs : \land\, msg.type = \text{“2a”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.coord = c \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, u \sqsubseteq msg.val
\end{array}
$$

$$
\begin{array}{l}
\quad \land \lor \land bA.vote[a][m] = none \\
\qquad\quad \land v = u \\
\qquad \lor \land AreCompatible(bA.vote[a][m],\ u) \\
\qquad\quad \land v = bA.vote[a][m] \sqcup u \\
\land bA' = [bA \text{ EXCEPT } !.vote[a][m] = v,\ !.mbal[a] = m] \\
\land msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto m,\ acc \mapsto a,\ val \mapsto v]\} \\
\land \text{UNCHANGED } \langle propCmd,\ learned,\ dMaxTried \rangle
\end{array}
$$

Action $Phase2bFast(a,\ C)$ is executed by acceptor a to accept command C, comming directly from the proposers (fast accept).

It implements the $AbstractMCPaxos!FastVote(a,\ C)$ action.

$$
\begin{array}{l}
Phase2bFast(a,\ C) \triangleq \\
\quad \land C \in propCmd \\
\quad \land IsFast(bA.mbal[a]) \\
\quad \land bA.vote[a][bA.mbal[a]] \neq none \\
\quad \land bA' = [bA \text{ EXCEPT } !.vote[a][bA.mbal[a]] = @ \bullet C] \\
\quad \land msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto bA.mbal[a],\ acc \mapsto a, \\
\qquad\qquad\qquad\qquad val \mapsto bA'.vote[a][bA.mbal[a]]]\} \\
\quad \land \text{UNCHANGED } \langle propCmd,\ learned,\ dMaxTried \rangle
\end{array}
$$

Action $Learn(l,\ v)$ executed by learner l to learn c-struct v.

It implements the $AbstractMCPaxos!AbstractLearn(a,\ v)$ action.

$$
\begin{array}{l}
Learn(l,\ v) \triangleq \\
\quad \land \exists\, m \in BalNum : \\
\qquad \exists\, Q \in Quorum(m) : \\
\qquad\quad \forall\, a \ \in Q : \exists\, msg \in msgs : \land msg.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land msg.bal \ = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land msg.acc \ = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land v \sqsubseteq msg.val \\
\quad \land learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v] \\
\quad \land \text{UNCHANGED } \langle propCmd,\ bA,\ dMaxTried,\ msgs \rangle
\end{array}
$$

Full Specification

Init defines the initial state.

$$
\begin{array}{l}
Init \ \triangleq \ \land propCmd = \{\} \\
\qquad\quad \land learned \quad = [l \in Learner \mapsto Bottom] \\
\qquad\quad \land bA = [vote \mapsto \\
\qquad\qquad\qquad [a \in Acceptor \mapsto
\end{array}
$$

$$[m \in BalNum \mapsto \text{IF } m = Zero \text{ THEN } Bottom \text{ ELSE } none]],$$
$$mbal \mapsto [a \in Acceptor \mapsto Zero]]$$
$$\land\ dMaxTried = [c \in Coord \mapsto [m \in BalNum \mapsto \text{IF } m = Zero \text{ THEN } Bottom \text{ ELSE } none]]$$
$$\land\ msgs = \{\}$$

Next defines how action are combined to generate the next states.

$$\begin{aligned} Next \triangleq\ & \lor \exists\, C \in Cmd : Propose(C) \\ & \lor \exists\, m \in BalNum,\ c \in Coord : Phase1a(c,\ m) \\ & \lor \exists\, m \in BalNum,\ v \in CStruct,\ c \in Coord : \\ & \qquad Phase2Start(c,\ m,\ v) \\ & \lor \exists\, m \in BalNum,\ s \in Seq(Cmd),\ c \in Coord : \\ & \qquad Phase2aClassic(c,\ m,\ s) \\ & \lor \exists\, a \in Acceptor,\ m \in BalNum : Phase1b(a,\ m) \\ & \lor \exists\, m \in BalNum,\ a \in Acceptor,\ v \in CStruct : \\ & \qquad Phase2bClassic(a,\ m,\ v) \\ & \lor \exists\, a \in Acceptor,\ C \in Cmd : Phase2bFast(a,\ C) \\ & \lor \exists\, l \in Learner,\ v \in CStruct : Learn(l,\ v) \end{aligned}$$

Spec is defined as the complete specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle propCmd,\ bA,\ learned,\ dMaxTried,\ msgs\rangle}$$

The following theorem asserts the invariance of *TypeInv*

THEOREM $Spec \Rightarrow \Box\, TypeInv$

The following theorem implies that there is a refinement mapping from *DistAbsMCPaxos* to *AbstractMCPaxos*. Therefore, *DistAbsMCPaxos* implements *AbstractMCPaxos*. Because *GeneralConsensus* is implemented by *AbstractMCPaxos*, *DistAbsMCPaxos* also implements *GeneralConsensus*

$$AB \triangleq \text{INSTANCE } AbstractMCPaxos \text{ WITH } maxTried \leftarrow MaxTried$$
$$GC \triangleq \text{INSTANCE } GeneralConsensus$$

THEOREM $Spec \Rightarrow AB!Spec$
THEOREM $Spec \Rightarrow GC!Spec$

B.4 Basic Multicoordinated Paxos

This module specifies the Multicoordinated Paxos algorithm as presented in Section 3.2.

MODULE *DistMCPaxos*

EXTENDS *PaxosConstants*

VARIABLES $crnd$, $cval$, rnd, $vrnd$, $vval$, $learned$, $msgs$
CONSTANT *Proposer*

$$cVars \triangleq \langle crnd, cval \rangle$$
$$aVars \triangleq \langle rnd, vrnd, vval \rangle$$

$$\begin{aligned} Msg \triangleq\ & [type : \{\text{“propose”}\},\ cmd : Cmd] \cup \\ & [type : \{\text{“1a”}\},\ bal : BalNum] \cup \\ & [type : \{\text{“1b”}\},\ bal : BalNum,\ vval\ : CStruct, \\ & \qquad\qquad vrnd : BalNum,\ acc : Acceptor] \cup \\ & [type : \{\text{“2a”}\},\ bal : BalNum,\ val : CStruct,\ coord : Coord] \cup \\ & [type : \{\text{“2b”}\},\ bal : BalNum,\ val : CStruct,\ acc : Acceptor] \end{aligned}$$

$$\begin{aligned} TypeInv \triangleq\ & \land\ crnd \in [Coord \to BalNum] \\ & \land\ cval\ \in [Coord \to CStruct] \\ & \land\ rnd\ \ \in [Acceptor \to BalNum] \\ & \land\ vrnd \in [Acceptor \to BalNum] \\ & \land\ vval\ \in [Acceptor \to CStruct] \\ & \land\ learned \in [Learner \to CStruct] \\ & \land\ msgs \subseteq Msg \end{aligned}$$

Actions

Action $Send(msg)$ implements the sending of message msg.

$$Send(msg) \triangleq msgs' = msgs \cup \{msg\}$$

$DistProvedSafe(Q, 1bMsg)$ below is the TLA$^+$ version of the $ProvedSafe(Q, 1bMsg)$ function presented in Section 3.3.

$$\begin{aligned} & DistProvedSafe(Q, 1bMsg) \triangleq \\ & \quad \text{LET} \\ & \qquad vals(S) \triangleq \{1bMsg[a].vval\ : a \in S\} \\ & \qquad vrnds \triangleq \{1bMsg[a].vrnd : a \in Q\} \\ & \qquad k \triangleq Max(vrnds) \\ & \qquad kacceptors \triangleq \{a \in Q : 1bMsg[a].vrnd = k\} \\ & \qquad QinterR \triangleq \{Q \cap R : R \in Quorum(k)\} \end{aligned}$$

$$
\begin{array}{l}
\quad QinterRAtk \triangleq \{S \in QinterR \quad : S \subseteq kacceptors\} \\
\quad Gamma \quad \triangleq \{GLB(vals(inter)) : inter \in QinterRAtk\} \\
\text{IN} \\
\quad \text{IF } QinterRAtk = \{\} \text{ THEN } vals(kacceptors) \\
\qquad\qquad\qquad\qquad \text{ELSE } \{LUB(Gamma)\}
\end{array}
$$

The following actions are the same as those presented in Section 3.3, just translated into TLA$^+$.

Propose$(p,\ C)$

$$
\begin{array}{l}
Propose(p,\ C) \triangleq \\
\quad \land \quad Send([type \mapsto \text{“propose”},\ cmd \mapsto C]) \\
\quad \land \quad \text{UNCHANGED } \langle cVars,\ aVars,\ learned \rangle
\end{array}
$$

*Phase*1*a*$(c,\ m)$

$$
\begin{array}{l}
Phase1a(c,\ m) \triangleq \\
\quad \land\ c \in \text{UNION } CoordQuorum(m) \\
\quad \land\ crnd[c] \prec m \\
\quad \land\ Send([type \mapsto \text{“1a”},\ bal \mapsto m]) \\
\quad \land\ \text{UNCHANGED } \langle cVars,\ aVars,\ learned \rangle
\end{array}
$$

*Phase*1*b*$(a,\ m)$

$$
\begin{array}{l}
Phase1b(a,\ m) \triangleq \\
\quad \land\ rnd[a] \prec m \\
\quad \land\ [type \mapsto \text{“1a”},\ bal \mapsto m] \in msgs \\
\quad \land\ rnd' = [rnd \text{ EXCEPT } ![a] = m] \\
\quad \land\ Send([type \mapsto \text{“1b”},\ bal \mapsto m,\ vval \mapsto vval[a], \\
\qquad\qquad\qquad\qquad vrnd \mapsto vrnd[a],\ acc \mapsto a]) \\
\quad \land\ \text{UNCHANGED } \langle cVars,\ vrnd,\ vval,\ learned \rangle
\end{array}
$$

*Phase*2*Start*$(c,\ m)$

$$
\begin{array}{l}
Phase2Start(c,\ m) \triangleq \\
\quad \land\ crnd[c] \prec m \\
\quad \land\ \exists\, Q \in Quorum(m) : \\
\qquad \land\ \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.bal\ \ = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.acc\ \ = a \\
\qquad \land\ \text{LET } 1bMsg \triangleq [a \in Q \mapsto \text{CHOOSE } msg \in msgs : \land\ msg.type = \text{“1b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.bal\ \ = m
\end{array}
$$

$\land\ msg.acc\ = a]$

IN $\exists\, w \in DistProvedSafe(Q,\, 1bMsg) :$
$\land\ crnd' = [crnd$ EXCEPT $![c] = m]$
$\land\ cval'\ = [cval$ EXCEPT $![c]\ = w]$
$\land\ Send([type \mapsto$ "2a", $bal \mapsto m,\ val \mapsto w,\ coord \mapsto c])$
$\land$ UNCHANGED $\langle aVars,\ learned\rangle$

Phase2aClassic(c)

$Phase2aClassic(c)\ \triangleq$
$\exists\, msg \in msgs :$
$\land\ msg.type =$ "propose"
$\land\ cval' = [cval$ EXCEPT $![c] = @ \bullet msg.cmd]$
$\land\ Send([type \mapsto$ "2a", $bal \mapsto crnd[c],\ val \mapsto cval'[c],\ coord \mapsto c])$
$\land$ UNCHANGED $\langle crnd,\ aVars,\ learned\rangle$

Phase2bClassic(a, m)

$Phase2bClassic(a,\ m)\ \triangleq$
$\land\ rnd[a] \preceq m$
$\land \exists\, L \in CoordQuorum(m) :$
$\land \forall\, c \in L : \exists\, msg \in msgs :\ \land\ msg.type =$ "2a"
$\land\ msg.bal = m$
$\land\ msg.coord = c$
$\land$ LET
$A2aMsg(c)\ \triangleq$ CHOOSE $msg \in msgs :\ \land\ msg.type =$ "2a"
$\land\ msg.bal = m$
$\land\ msg.coord = c$
$L2aMsgs\ \triangleq\ \{A2aMsg(c) : c \in L\}$
$L2aVals\ \triangleq\ \{msg.val : msg\ \in L2aMsgs\}$
IN
$\land\ \lor\ \land\ vrnd[a] \prec m$
$\land\ vval' = [vval$ EXCEPT $![a] = GLB(L2aVals)]$
$\lor\ \land\ vrnd[a] = m$
$\land\ AreCompatible(vval[a],\ GLB(L2aVals))$
$\land\ vval' = [vval$ EXCEPT $![a] = @ \sqcup GLB(L2aVals)]$
$\land\ rnd'\ = [rnd\ $ EXCEPT $![a] = m]$
$\land\ vrnd' = [vrnd$ EXCEPT $![a] = m]$
$\land\ Send([type \mapsto$ "2b", $bal \mapsto m,\ val \mapsto vval'[a],\ acc \mapsto a])$
$\land$ UNCHANGED $\langle cVars,\ learned\rangle$

Phase2bFast(a)

$Phase2bFast(a) \triangleq$
$\land\ IsFast(rnd[a])$
$\land\ rnd[a] = vrnd[a]$
$\land\ \exists\, msg \in msgs :$
$\land\ msg.type = \text{“2b”}$
$\land\ vval' = [vval \text{ EXCEPT } ![a] = @ \bullet msg.cmd]$
$\land\ Send([type \mapsto \text{“2b”},\ bal \mapsto rnd[a],\ val \mapsto vval'[a],\ acc \mapsto a])$
$\land\ \text{UNCHANGED } \langle cVars,\ rnd,\ vrnd,\ learned\rangle$

Learn(l)

$Learn(l) \triangleq$
$\exists\, m \in BalNum :$
$\exists\, Q \in Quorum(m) :$
$\land\ \forall\, a \in Q : \exists\, msg \in msgs :\ \land\ msg.type = \text{“2b”}$
$\land\ msg.bal = m$
$\land\ msg.acc = a$
$\land\ \text{LET}$
$A2bMsg(a) \triangleq \text{CHOOSE } msg \in msgs :\ \land\ msg.type = \text{“2b”}$
$\land\ msg.bal = m$
$\land\ msg.acc \in Q$
$Q2bMsgs \triangleq \{A2bMsg(a) : a \in Q\}$
$Q2bVals \triangleq \{msg.val : msg \in Q2bMsgs\}$
IN
$\land\ learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup GLB(Q2bVals)]$
$\land\ \text{UNCHANGED } \langle cVars,\ aVars,\ msgs\rangle$

Full Specification

Init defines the initial state.

$Init \triangleq\ \land\ learned = [l \in Learner \mapsto Bottom]$
$\land\ crnd = [c \in Coord \mapsto Zero]$
$\land\ cval = [c \in Coord \mapsto Bottom]$
$\land\ rnd = [a \in Acceptor \mapsto Zero]$
$\land\ vrnd = [a \in Acceptor \mapsto Zero]$
$\land\ vval = [a \in Acceptor \mapsto Bottom]$
$\land\ learned = [l \in Learner \mapsto Bottom]$
$\land\ msgs = \{\}$

Next defines how action are combined to generate the next states.

$$
\begin{aligned}
Next \triangleq\ & \lor \exists\, p \in Proposer,\ C \in Cmd\ : Propose(p,\ C) \\
& \lor \exists\, c \in Coord,\ m \in BalNum : Phase1a(c,\ m) \\
& \lor \exists\, c \in Coord,\ m \in BalNum : Phase2Start(c,\ m) \\
& \lor \exists\, c \in Coord : Phase2aClassic(c) \\
& \lor \exists\, a \in Acceptor,\ m \in BalNum : Phase1b(a,\ m) \\
& \lor \exists\, a \in Acceptor,\ m \in BalNum : Phase2bClassic(a,\ m) \\
& \lor \exists\, a \in Acceptor : Phase2bFast(a) \\
& \lor \exists\, l\ \in Learner : Learn(l)
\end{aligned}
$$

Spec is defined as the complete specification.

$Spec \triangleq Init \land \Box[Next]_{\langle cVars,\, aVars,\, learned,\, msgs\rangle}$

The following theorem asserts the invariance of *TypeInv*.

THEOREM $Spec \Rightarrow \Box\, TypeInv$

The following theorem asserts the *DistMCPaxos* implements *GeneralConsensus*

$PropCmd \triangleq \{m.cmd : m \in \{mm \in msgs : mm.type = \text{“propose”}\}\}$

$GC \triangleq$ INSTANCE *GeneralConsensus* WITH $propCmd \leftarrow PropCmd$

THEOREM $Spec \Rightarrow GC!Spec$

B.5 Complete Multicoordinated Paxos

This module specifies the Multicoordinated Paxos algorithm without dependencies, also specifying the collision detection mechanisms and a simplified version of the mechanism presented in Section 4.4 to reduce the number of disk writes.

MODULE *MultiCoordPaxos*

The module imports two standard modules. Module *Naturals* defines the set *Nat* of naturals and the ordinary arithmetic operators; module *FiniteSets* defines *IsFiniteSet*(s) to be true iff S is a finite set and defines *Cardinality*(S) to be the number of elements in S, if s is finite.

EXTENDS *Naturals*, *FiniteSets*, *CStructs*

Constants

The next statement declares the specification's constant parameters, which have the following meanings:

Acceptor	the set of acceptors.
Learner	the set of learners.
FastNum	the set of fast round numbers.
Quorum(i)	the set of i-quorums.
Coord	the set of coordinators.
CoordQuorum(i)	the set of coordinator quorums of round i.

CONSTANTS $Acceptor$, $Learner$, $FastNum$, $Quorum(\_)$, $Coord$, $CoordQuorum(\_)$

PosNat is defined to be the set of positive integers.

$$PosNat \triangleq Nat \setminus \{0\}$$

RNum is the set of round numbers. A round number is composed of two parts: the incarnation number and the sequence number.

$$RNum \triangleq Nat \times PosNat$$

A round number with sequence number 0 is not a valid round number, but it is used to represent some agents' states. *RType* is defined to represent such a set.

$$RType \triangleq Nat \times Nat$$

We must define a precedence relation between round numbers. We define $\prec$ to represent the relation that round i precedes round j iff i has a lower incarnation number than j or they have the same incarnation number but i has a lower sequence number than j. We also define $\preceq$, $\succ$, and $\succeq$ accordingly.

$$i \prec j \triangleq \text{IF } i[1] < j[1] \text{ THEN TRUE ELSE } i[2] < j[2]$$
$$i \preceq j \triangleq i \prec j \vee i = j$$
$$i \succ j \triangleq j \prec i$$
$$i \succeq j \triangleq j \preceq i$$

Max(S) is defined to be the maximum of a finite set S of Round Numbers.

$$Max(S) \triangleq \text{CHOOSE } i \in S : \forall j \in S : i \succeq j$$

the following statement asserts the assumption that *FastNum* is a set of round numbers.

ASSUME $FastNum \subseteq RNum$

ClassicNum is defined to be the set of classic round numbers.

$$ClassicNum \triangleq RNum \setminus FastNum$$

FairNum(c) is defined to be the set of classic round numbers for which coordinator c is, itself, a coordinator quorum.

$$FairNum(c) \triangleq \{i \in ClassicNum : \{c\} \in CoordQuorum(i)\}$$

The following assumption asserts that the set of acceptors is finite. It is needed to ensure progress.

ASSUME $IsFiniteSet(Acceptor)$

The following asserts the assumptions that $Quorum(i)$ is a set of sets of acceptors, for every round number i, and that the *Quorum* Requirement holds.

ASSUME $\forall\, i \in RNum :$
$\quad \land\ Quorum(i) \subseteq \text{SUBSET}\ Acceptor$
$\quad \land\ \forall\, j \in RNum :$
$\quad\quad \land\ \forall\, Q \in Quorum(i),\ R \in Quorum(j) : Q \cap R \neq \{\}$
$\quad\quad \land\ (j \in FastNum) \Rightarrow$
$\quad\quad\quad \forall\, Q \in Quorum(i) : \forall\, R1,\ R2 \in Quorum(j) :$
$\quad\quad\quad\quad Q \cap R1 \cap R2 \neq \{\}$

The following asserts the assumptions that $CoordQuorum(i)$ is a set of sets of coordinators, for every round number i, and that every coordinator is, itself, a coordinator quorum of infinitely many classic rounds for every possible incarnation.

ASSUME $\land\ \forall\, i \in RNum : CoordQuorum(i) \subseteq \text{SUBSET}\ Coord$
$\quad \land\ \forall\, c \in Coord,\ i \in RType :$
$\quad\quad \exists\, j \in PosNat :\ \land\ j > i[2]$
$\quad\quad\quad\quad\quad\quad\quad\quad \land\ \langle i[1],\ j\rangle \in FairNum(c)$

The following asserts the assumption that coordinator quorums of the same round should always intersect.

ASSUME $\forall\, i \in RNum : \forall\, Q,\ R \in CoordQuorum(i) : Q \cap R \neq \{\}$

Message is defined to be the set of all possible messages. A message is a record having a *type* field indicating what message it is, and a *rnd* field indicating the round number. What other fields, if any, a message has depends on its type.

$$
\begin{array}{lll}
Message \triangleq & [type : \{\text{“phase1a”}\},\ rnd : RNum] \\
\cup & [type : \{\text{“phase1b”}\},\ rnd : RNum,\ vval : CStruct, \\
 & \qquad\qquad\qquad\quad vrnd : RType,\ acc : Acceptor] \\
\cup & [type : \{\text{“phase2a”}\},\ rnd : RNum,\ val : CStruct, \\
 & \qquad\qquad\qquad\quad coord : Coord] \\
\cup & [type : \{\text{“phase2b”}\},\ rnd : RNum,\ val : CStruct, \\
 & \qquad\qquad\qquad\quad acc\ : Acceptor]
\end{array}
$$

Variables and State Predicates

The following statement declares the specification's variables.

VARIABLES rnd, $vrnd$, $vval$, $crnd$, $cval$, $amLeader$, $sentMsg$, $proposed$, $learned$, $goodSet$

Defining the following tuples of variables makes it more convenient to state which variables are left unchanged by the actions.

$aVars \triangleq \langle rnd,\ vrnd,\ vval \rangle$ Acceptor variables

$cVars \triangleq \langle crnd,\ cval \rangle$ Coordinator variables

$oVars \triangleq \langle amLeader,\ proposed,\ learned,\ goodSet \rangle$ Most other variables

$vars \triangleq \langle aVars,\ cVars,\ oVars,\ sentMsg \rangle$ All variables

TypeOK is the type-correctness invariant, asserting that the value of each variable is an element of the proper set (its "type"). Type correctness of the specification means that *TypeOK* is an invariant—— that is, it is true in every state of every behavior allowed by the specification.

$$
\begin{array}{l}
TypeOK \triangleq \\
\quad \land \quad rnd \in [Acceptor \to RType] \\
\quad \land \quad vrnd \in [Acceptor \to RType] \\
\quad \land \quad vval \in [Acceptor \to CStruct] \\
\quad \land \quad crnd \in [Coord \to RType] \\
\quad \land \quad cval \in [Coord \to CStruct \cup \{none\}] \\
\quad \land \quad amLeader \in [Coord \to \text{BOOLEAN}] \\
\quad \land \quad sentMsg \in \text{SUBSET } Message \\
\quad \land \quad proposed \in \text{SUBSET } Cmd \\
\quad \land \quad learned \in [Learner \to CStruct] \\
\quad \land \quad goodSet \subseteq Acceptor \cup Coord
\end{array}
$$

Init is the initial predicate that describes the initial values of all variables.

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ rnd = [a \in Acceptor \mapsto \langle 0,\ 0 \rangle] \\
\quad \land\ vrnd = [a \in Acceptor \mapsto \langle 0,\ 0 \rangle] \\
\quad \land\ vval = [a \in Acceptor \mapsto Bottom] \\
\quad \land\ crnd = [a \in Coord \mapsto \langle 0,\ 0 \rangle] \\
\quad \land\ cval = [c \in Coord \mapsto none] \\
\quad \land\ amLeader \in [Coord \to \text{BOOLEAN}] \\
\quad \land\ sentMsg = \{\} \\
\quad \land\ proposed = \{\} \\
\quad \land\ learned = [l \in Learner \mapsto Bottom] \\
\quad \land\ goodSet \in \text{SUBSET } (Acceptor \cup Coord)
\end{array}
$$

Action Definitions

Send(m) describes the state change that represents the sending of a message m. It is used as a conjunt in defining the algorithm actions.

$Send(msg) \triangleq sentMsg' = sentMsg \cup \{msg\}$

Coordinator Actions

Action $Phase1a(c, i)$ specifies the execution of phase $1a$ of round i by coordinator c. Different from the previous specifications, this action changes *crnd* and *cval*. This is done for liveness, to prevent a coordinator from continuously starting new rounds. It could be done by adding a new variable but we just thought that this way was easier and compliant with other *Paxos* specifications.

$$
\begin{array}{l}
Phase1a(i,\ c) \triangleq \\
\quad \land\ amLeader[c] \\
\quad \land\ c \in \text{UNION}\ CoordQuorum(i) \\
\quad \land\ crnd[c] \prec i \\
\quad \land\ \lor\ crnd[c] = \langle i[1],\ 0\rangle \\
\qquad \lor\ \exists\, m \in sentMsg :\ \land\ crnd[c] \prec m.rnd \\
\qquad\qquad\qquad\qquad\ \land\ m.rnd[1] = i[1] \\
\qquad\qquad\qquad\qquad\ \land\ m.rnd \prec i \\
\qquad \lor\ \land\ crnd[c] \notin FairNum(c) \\
\qquad\quad\ \land\ i[1] = crnd[c][1] \\
\quad \land\ crnd' = [crnd\ \text{EXCEPT}\ ![c] = i] \\
\quad \land\ cval' = [cval\ \text{EXCEPT}\ ![c] = none] \\
\quad \land\ Send([type \mapsto \text{“phase1a”},\ rnd \mapsto i]) \\
\quad \land\ \text{UNCHANGED}\ \langle aVars,\ oVars\rangle
\end{array}
$$

Reasons for executing $Phase1a$:

1 - Did not do anything in this in-

2 - Some round interfered with round $crnd[c]$

3 - Round $crnd[c]$ might have collisions and cannot ensure liveness in the presence of failures

MsgsFrom$(Q, i, phase)$ is defined to be the set of messages in *sentMsg* of type *phase* (which may equal “phase1b” or “phase2b”) sent in round i by the acceptors in the set Q.

$$
\begin{array}{l}
MsgsFrom(Q,\ i,\ phase) \triangleq \\
\quad \{m \in sentMsg : (m.type = phase) \land (m.acc \in Q) \land (m.rnd = i)\}
\end{array}
$$

If M is the set of round i phase $1b$ messages sent by the acceptors in a quorum Q, then *IsPickableVal*(Q, i, M, v) is true according to the following rule, easily derived from the definition of *ProvedSafe* in the paper. It allows the coordinator to send the value v in a phase $2a$ message for round i.

$$
\begin{array}{l}
IsPickableVal(Q,\ i,\ M,\ v) \triangleq \\
\quad \text{LET}\ vr(a) \triangleq (\text{CHOOSE}\ m \in M : m.acc = a).vrnd \\
\qquad\ vv(a) \triangleq (\text{CHOOSE}\ m \in M : m.acc = a).vval \\
\qquad\ k \triangleq Max(\{vr(a) : a \in Q\}) \\
\qquad\ RS \triangleq \{R \in Quorum(k) : \forall\, a \in Q \cap R : vr(a) = k\} \\
\qquad\ g(R) \triangleq GLB(\{vv(a) : a \in R \cap Q\}) \\
\qquad\ G \triangleq \{g(R) : R \in RS\} \\
\qquad\ PrSafe \triangleq \\
\qquad\quad \text{IF}\ RS = \{\}\ \text{THEN}\ \{vv(a) : a \in \{b \in Q : vr(b) = k\}\}
\end{array}
$$

ELSE $\{LUB(G)\}$

IN $\exists\, w \in PrSafe,\, s \in Seq(proposed) : v = w \mathbin{**} s$

*Phase*2*Start*$(i,\, c,\, v)$ specifies the first execution of *phase*2*a* in round i by coordinator c.

$Phase2Start(i,\, c,\, v) \triangleq$
has executed *phase*1*a*, but not *phase*2*a*, or another coordinator executed *phase*1*a*.

$$
\begin{array}{l}
\land\ \lor\ \land\ crnd[c] = i \\
\qquad\quad \land\ cval[c] = none \\
\qquad \lor\ crnd[c] \prec i \\
\land\ \exists\, Q \in Quorum(i) : \\
\qquad \land\ \forall\, a \in Q : \exists\, m \in MsgsFrom(Q,\, i,\, \text{“phase1b”}) : m.acc = a \\
\qquad \land\ IsPickableVal(Q,\, i,\, MsgsFrom(Q,\, i,\, \text{“phase1b”}),\, v) \\
\land\ cval' = [cval \text{ EXCEPT } ![c] = v] \\
\land\ crnd' = [crnd \text{ EXCEPT } ![c] = i] \\
\land\ Send([type \mapsto \text{“phase2a”},\, rnd \mapsto i,\, val \mapsto v,\, coord \mapsto c]) \\
\land\ \text{UNCHANGED } \langle aVars,\, oVars \rangle
\end{array}
$$

*Phase*2*a*$(i,\, c,\, v)$ specifies other executions of *phase*2*a* in round i by coordinator c.

$Phase2a(i,\, c,\, v) \triangleq$
has executed *Phase*2*Start*.

$$
\begin{array}{l}
\land\ \ crnd[c] = i \\
\land\ \ cval[c] \neq none \\
\land\ \ \exists\, C \in proposed : v = cval[c] \mathbin{**} C \\
\land\ cval' = [cval \text{ EXCEPT } ![c] = v] \\
\land\ Send([type \mapsto \text{“phase2a”},\, rnd \mapsto i,\, val \mapsto v,\, coord \mapsto c]) \\
\land\ \text{UNCHANGED } \langle crnd,\, aVars,\, oVars \rangle
\end{array}
$$

NextRound(i) is the round number following i in the same incarnation.

$NextRound(i) \triangleq [i \text{ EXCEPT } ![2] = @ + 1]$

*NextRoundP*1*b*$(Q,\, i)$ is the set of phase 1*b* messages for round *NextRound*(i) sent by acceptors in Q.

$NextRoundP1b(Q,\, i) \triangleq MsgsFrom(Q,\, NextRound(i),\, \text{“phase1b”})$

Action *CoordinatedRecovery*$(i,\, c,\, v)$ specifies our variation of coordinated recovery. With this action, coordinator c attempts to recover from a collision in round i by sending round *NextRound*(i) phase 2*a* messages for the value v. To ensure liveness, *NextRound*(i) should be a fair round, but this is not a requirement for correctness.

$CoordinatedRecovery(i,\, c,\, v) \triangleq$

$$
\begin{array}{ll}
\text{LET } j \triangleq NextRound(i) & \\
\text{IN} & \land\ crnd[c] \prec j \\
& \land\ \exists\, Q \in Quorum(j) : \\
& \qquad \land\ \forall\, a \in Q : \exists\, m \in NextRoundP1b(Q,\, i) : m.acc = a
\end{array}
$$

$\land$ $IsPickableVal(Q,\ j,\ NextRoundP1b(Q,\ i),\ v)$
$\land$ $cval' = [cval$ EXCEPT $![c] = v]$
$\land$ $crnd' = [crnd$ EXCEPT $![c] = j]$
$\land$ $Send([type \mapsto$ "phase2a", $rnd \mapsto j,\ val \mapsto v,\ coord \mapsto c])$
$\land$ UNCHANGED $\langle aVars,\ oVars\rangle$

coordLastMsg(c) is defined to be the last message that coordinator c sent, if $crnd[c] \succ \langle 0,\ 0\rangle$.

$coordLastMsg(c) \triangleq$
IF $cval[c] = none$
THEN $[type \mapsto$ "phase1a", $rnd \mapsto crnd[c]]$
ELSE $[type \mapsto$ "phase2a", $rnd \mapsto crnd[c],\ val \mapsto cval[c],\ coord \mapsto c]$

In action *CoordRetransmit*(c), coordinator c retransmits the last message it sent. This action is a stuttering action (meaning it does not change the value of any variable, so it is a no-op) if that message is still in *sentMsg*. However, this action is needed because c might have failed after first sending the message and subsequently have been repaired after the message was removed from *sentMsg*.

$CoordRetransmit(c) \triangleq$
$\land$ $crnd[c] \in RNum$
$\land$ $Send(coordLastMsg(c))$
$\land$ UNCHANGED $\langle aVars,\ cVars,\ amLeader,\ proposed,\ learned,\ goodSet\rangle$

CoordNext(c) is the next-state action of coordinator c—— that is, the disjunct of the algorithm's complete next-state action that represents actions of that coordinator.

$CoordNext(c) \triangleq$
$\lor \exists\, i \in RNum :\ \lor Phase1a(i,\ c)$
$\lor \exists\, v \in CStruct :\ \lor Phase2Start(i,\ c,\ v)$
$\lor Phase2a(i,\ c,\ v)$
$\lor CoordinatedRecovery(i,\ c,\ v)$
$\lor CoordRetransmit(c)$

Acceptor Actions

Action $Phase1b(i,\ a)$ specifies the execution of phase $1b$ for round i by acceptor a.

$Phase1b(i,\ a) \triangleq$
$\land$ $rnd[a] \prec i$
$\land$ $[type \mapsto$ "phase1a", $rnd \mapsto i] \in sentMsg$
$\land$ $rnd' = [rnd$ EXCEPT $![a] = i]$
$\land$ $Send([type \mapsto$ "phase1b", $rnd \mapsto i,\ vrnd \mapsto vrnd[a],\ vval \mapsto vval[a],$
$acc \mapsto a])$
$\land$ UNCHANGED $\langle cVars,\ oVars,\ vrnd,\ vval\rangle$

MsgsFromCoordQuorum(Q, i, *phase*) is defined to be the set of messages in *sentMsg* of type *phase* (which may equal "phase1a" or "phase2a") sent in round i by the coordinators in the set Q.

$$MsgsFromCoordQuorum(Q,\ r,\ phase) \triangleq$$
$$\{m \in sentMsg : (m.type = phase) \land (m.coord \in Q) \land (m.rnd = r)\}$$

Action *Phase*2*b*(i, a, v) specifies the execution of phase 2b for round i by acceptor a, upon receipt of either a phase 2a message or a proposal (for a fast round) with value v. This action actually implements actions *Phase*2*bClassic* and *Phase*2*bFast* of the previous specifications

$$\begin{array}{ll}
Phase2b(i,\ a,\ v) \triangleq & \\
\quad \land & rnd[a] \preceq i \\
\quad \land & \lor\ vrnd[a] \prec i \\
 & \lor\ vval[a] \sqsubset v \\
\quad \land & \lor \exists\, Q \in CoordQuorum(i),\ u \in CStruct : \\
 & \quad \land \forall\, c \in Q : \exists\, m \in MsgsFromCoordQuorum(Q,\ i,\ \text{"phase2a"}) : \\
 & \qquad \land\ m.coord = c \\
 & \qquad \land\ m.val \sqsubseteq u \\
 & \quad \land\ \lor\ \land\ vrnd[a] \prec i \\
 & \qquad \land\ v = u \\
 & \qquad \lor\ \land\ vrnd[a] = i \\
 & \qquad \land\ AreCompatible(vval[a],\ u) \\
 & \qquad \land\ v = vval[a] \sqcup u \\
 & \quad \lor\ \land\ i \in FastNum \\
 & \qquad \land\ vrnd[a] = i \\
 & \qquad \land \exists\, C \in proposed : v = vval[a] **C \quad \text{Sent in classic } 2a. \\
\quad \land & rnd' = [rnd \ \text{EXCEPT}\ ![a] = i] \\
\quad \land & vrnd' = [vrnd\ \text{EXCEPT}\ ![a] = i] \\
\quad \land & vval' = [vval\ \text{EXCEPT}\ ![a] = v] \\
\quad \land & Send([type \mapsto \text{"phase2b"},\ rnd \mapsto i,\ val \mapsto v,\ acc \mapsto a]) \\
\quad \land & \text{UNCHANGED}\ \langle cVars,\ oVars\rangle
\end{array}$$

Action *CollisionDetection*(i, a) specifies the action acceptor a must take when it detects a collision on the values proposed by a coordinator quorum or on the values accepted by an acceptor quorum for round i. In such a case, acceptor a sends a *phase*1b message for round $i+1$.

$$\begin{array}{ll}
CollisionDetection(i,\ a) \triangleq & \\
\quad \land & rnd[a] \preceq i \\
\quad \land & \lor \exists\, Q \in CoordQuorum(i) : \quad \text{collision in a multicoordinated round} \\
 & \quad \exists\, m1,\ m2 \in MsgsFromCoordQuorum(Q,\ i,\ \text{"phase2a"}) : \\
 & \qquad \land\ m1.coord \neq m2.coord \\
 & \qquad \land \neg AreCompatible(m1.val,\ m2.val)
\end{array}$$

$\lor \exists\, Q \in Quorum(i) :$ collision in a fast round
$\quad \exists\, m1,\, m2 \in MsgsFrom(Q,\, i,\, \text{“phase2b”}) :$
$\quad\quad \land\, m1.acc \neq m2.acc$
$\quad\quad \land\, \neg AreCompatible(m1.val,\, m2.val)$
$\land\, rnd' = [rnd \text{ EXCEPT } ![a] = i]$
$\land\, Send([type \mapsto \text{“phase1b”},\, rnd \mapsto NextRound(i),\, vrnd \mapsto vrnd[a],$
$\quad vval \mapsto vval[a],\, acc \mapsto a])$
$\land$ UNCHANGED $\langle cVars,\, oVars,\, vrnd,\, vval\rangle$

Action *UncoordinatedRecovery*$(i,\, a,\, v)$ specifies our variation of uncoordinated recovery. With this action, acceptor a attempts to recover from a collision in round i by sending a phase $2b$ message for round *NextRound*(i) with value v.

$UncoordinatedRecovery(i,\, a,\, v) \triangleq$
LET $j \triangleq NextRound(i)$
IN $\land\, j \in FastNum$
$\quad \land\, rnd[a] \preceq i$
$\quad \land \exists\, Q \in Quorum(j) :$
$\quad\quad \land \forall\, b \in Q : \exists\, m \in NextRoundP1b(Q,\, i) : m.acc = b$
$\quad\quad \land\, IsPickableVal(Q,\, j,\, NextRoundP1b(Q,\, i),\, v)$
$\quad \land\, rnd' = [rnd \text{ EXCEPT } ![a] = j]$
$\quad \land\, vrnd' = [vrnd \text{ EXCEPT } ![a] = j]$
$\quad \land\, vval' = [vval \text{ EXCEPT } ![a] = v]$
$\quad \land\, Send([type \mapsto \text{“phase2b”},\, rnd \mapsto j,\, val \mapsto v,\, acc \mapsto a])$
$\quad \land$ UNCHANGED $\langle cVars,\, oVars\rangle$

accLastMsg(a) is defined to be the last message sent by acceptor a, if $rnd[a] \succ \langle 0,\, 0\rangle$

$accLastMsg(a) \triangleq$
IF $vrnd[a] \prec rnd[a]$
THEN $[type \mapsto \text{“phase1b”},\, rnd \mapsto rnd[a],\, vrnd \mapsto vrnd[a],$
$\quad vval \mapsto vval[a],\, acc \mapsto a]$
ELSE $[type \mapsto \text{“phase2b”},\, rnd \mapsto rnd[a],\, val \mapsto vval[a],$
$\quad acc \mapsto a]$

In action *AcceptorRetransmit*(a) acceptor a retransmits the last message it sent.

$AcceptorRetransmit(a) \triangleq$
$\land\, rnd[a] \in RNum$
$\land\, Send(accLastMsg(a))$
$\land$ UNCHANGED $\langle aVars,\, cVars,\, amLeader,\, proposed,\, learned,\, goodSet\rangle$

AcceptorNext(a) is the next-state action of acceptor a—— that is, the disjunct of the next-state action that represents actions of that acceptor.

$$
\begin{array}{l}
AcceptorNext(a) \triangleq \\
\quad \lor \exists\, i \in RNum : \lor Phase1b(i,\, a) \\
\qquad\qquad\qquad\qquad \lor \exists\, v \in CStruct : \lor Phase2b(i,\, a,\, v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor UncoordinatedRecovery(i,\, a,\, v) \\
\qquad\qquad\qquad\qquad \lor CollisionDetection(i,\, a) \\
\quad \lor AcceptorRetransmit(a)
\end{array}
$$

Other Actions

Action $Propose(v)$ represents the proposal of a value v by some proposer.

$$
\begin{array}{l}
Propose(v) \triangleq \\
\quad \land \quad proposed' = proposed \cup \{v\} \\
\quad \land \quad \text{UNCHANGED } \langle aVars,\, cVars,\, amLeader,\, sentMsg,\, learned,\, goodSet \rangle
\end{array}
$$

Action $Learn(l,\, v)$ represents the learning of a value v by learner l.

$$
\begin{array}{l}
Learn(l,\, v) \triangleq \\
\quad \land \exists\, i \in RNum : \\
\qquad \exists\, Q \in Quorum(i) : \\
\qquad\quad \forall\, a \in Q : \\
\qquad\qquad \exists\, m \in sentMsg : \land m.type = \text{“phase2b”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.rnd = i \\
\qquad\qquad\qquad\qquad\qquad\quad \land m.acc = a \\
\qquad\qquad\qquad\qquad\qquad\quad \land v \sqsubseteq m.val \\
\quad \land learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup \{v\}] \\
\quad \land \text{UNCHANGED } \langle aVars,\, cVars,\, amLeader,\, sentMsg,\, proposed,\, goodSet \rangle
\end{array}
$$

Action *LeaderSelection* allows an arbitrary change to the values of $amLeader[c]$, for all coordinators c. Since this action may be performed at any time, the specifiction makes no assumption about the outcome of leader selection. (However, progress is guaranteed only under an assumption about the values of $amLeader[c]$.)

$$
\begin{array}{l}
LeaderSelection \triangleq \\
\quad \land amLeader' \in [Coord \to \text{BOOLEAN }] \\
\quad \land \text{UNCHANGED } \langle aVars,\, cVars,\, sentMsg,\, proposed,\, learned,\, goodSet \rangle
\end{array}
$$

Action $Fail(a)$ specifies the failure of agent a.

$$
\begin{array}{l}
Fail(a) \triangleq \\
\quad \land a \in goodSet \\
\quad \land goodSet' = goodSet \setminus \{a\} \\
\quad \land \text{UNCHANGED } \langle aVars,\, cVars,\, amLeader,\, sentMsg,\, proposed,\, learned \rangle
\end{array}
$$

Repair(a) specifies the recovery of agent a. For simplicity, we model the loss of state in $crnd[a]$, $cval[a]$, or $rnd[a]$ during recovery.

$$
\begin{array}{l}
Repair(a) \triangleq \\
\quad \land\ a \notin goodSet \\
\quad \land\ goodSet' = goodSet \cup \{a\} \\
\quad \land\ \text{IF}\ a \in Coord \\
\qquad \text{THEN}\ \land\ crnd' = [crnd\ \text{EXCEPT}\ ![a][1] = @ + 1,\ ![a][2] = 0] \\
\qquad\qquad\ \ \land\ cval' = [cval\ \text{EXCEPT}\ ![a] = none] \\
\qquad \text{ELSE}\ \text{UNCHANGED}\ cVars \\
\quad \land\ \text{IF}\ a \in Acceptor \\
\qquad \text{THEN}\ \land\ rnd' = [rnd\ \text{EXCEPT}\ ![a][1] = @ + 1,\ ![a][2] = 1] \\
\qquad \text{ELSE}\ \text{UNCHANGED}\ rnd \\
\quad \land\ \text{UNCHANGED}\ \langle vrnd,\ vval,\ amLeader,\ sentMsg,\ proposed,\ learned\rangle
\end{array}
$$

Action *FailOrRepair* allows the failure or recovery of an agent a. Since this action may be performed at any time, the specification makes no assumption about which agents are good. (However, progress is guaranteed only under an assumption about the value of *goodSet*.)

$$
\begin{array}{l}
FailOrRepair \triangleq\ \exists\, a \in (Coord \cup Acceptor) : \\
\qquad\qquad\qquad \lor\ Fail(a) \\
\qquad\qquad\qquad \lor\ Repair(a)
\end{array}
$$

Action $LoseMsg(m)$ removes message m from *sentMsg*. It is always enabled unless m is the last message sent by an acceptor or coordinator in *goodSet*. Hence, the only assumption the specification makes about message loss is that the last message sent by an agent in *goodSet* is not lost. Because *sentMsg* includes messages in an agent's output buffer, this effectively means that a non-failed process always has the last message it sent in its output buffer, ready to be retransmitted.

$$
\begin{array}{l}
LoseMsg(m) \triangleq \\
\quad \land\ \neg \lor\ \land\ m.type = \text{“phase1a”} \\
\qquad\qquad\ \land\ \exists\, c \in \text{UNION}\ CoordQuorum(m.rnd) : \\
\qquad\qquad\qquad\ \land\ m = coordLastMsg(c) \\
\qquad\qquad\qquad\ \land\ c\ \ \in goodSet \\
\qquad\quad \lor\ \land\ m.type = \text{“phase2a”} \\
\qquad\qquad\ \land\ m = coordLastMsg(m.coord) \\
\qquad\qquad\ \land\ m.coord \in goodSet \\
\qquad\quad \lor\ \land\ m.type \in \{\text{“phase1b”},\ \text{“phase2b”}\} \\
\qquad\qquad\ \land\ m = accLastMsg(m.acc) \\
\qquad\qquad\ \land\ m.acc \in goodSet \\
\quad \land\ sentMsg' = sentMsg \setminus \{m\} \\
\quad \land\ \text{UNCHANGED}\ \langle aVars,\ cVars,\ amLeader,\ proposed,\ learned,\ goodSet\rangle
\end{array}
$$

Action *OtherAction* is the disjunction of all actions other than ones peformed by acceptors or coordinators, plus the *LeaderSelection* action (which represents leader-selection actions performed by the coordinators).

$$
\begin{aligned}
&\mathit{OtherAction} \triangleq \\
&\quad \lor \exists\, v \in \mathit{Cmd} : \mathit{Propose}(v) \\
&\quad \lor \exists\, v \in \mathit{CStruct},\, l \in \mathit{Learner} : \mathit{Learn}(l,\, v) \\
&\quad \lor \mathit{LeaderSelection} \\
&\quad \lor \mathit{FailOrRepair} \\
&\quad \lor \exists\, m \in \mathit{sentMsg} : \mathit{LoseMsg}(m)
\end{aligned}
$$

Next is the algorithm's complete next-state action.

$$
\begin{aligned}
&\mathit{Next} \triangleq \\
&\quad \lor \exists\, c \in \mathit{Coord} : \mathit{CoordNext}(c) \\
&\quad \lor \exists\, a \in \mathit{Acceptor} : \mathit{AcceptorNext}(a) \\
&\quad \lor \mathit{OtherAction}
\end{aligned}
$$

Formula *Spec* is the complete specification of the Multi-coordinated Paxos algorithm without fairness.

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$$

The following are the safety properties of Generalized Consensus.

$$
\begin{aligned}
\mathit{Nontriviality} &\triangleq \forall\, l \in \mathit{Learner} : \\
&\qquad \Box(\mathit{learned}[l] \in \mathit{Str}(\mathit{proposed})) \\
\mathit{Stability} &\triangleq \forall\, l \in \mathit{Learner},\, v \in \mathit{CStruct} : \\
&\qquad \Box((\mathit{learned}[l] = v) \Rightarrow \Box(v \sqsubseteq \mathit{learned}[l])) \\
\mathit{Consistency} &\triangleq \forall\, l1,\, l2 \in \mathit{Learner} : \\
&\qquad \Box\, \mathit{AreCompatible}(\mathit{learned}[l1],\, \mathit{learned}[l2])
\end{aligned}
$$

The following theorem asserts the correctness of the algorithm.

THEOREM $\mathit{Spec} \Rightarrow \Box(\mathit{TypeOK}) \land \mathit{Nontriviality} \land \mathit{Stability} \land \mathit{Consistency}$