Generalized Consensus and Paxos

Leslie Lamport

3 March 2004
revised 15 March 2005
corrected 28 April 2005

Microsoft Research Technical Report MSR-TR-2005-33

Abstract

Theoretician's Abstract

Consensus has been regarded as the fundamental problem that must be solved to implement a fault-tolerant distributed system. However, only a weaker problem than traditional consensus need be solved. We generalize the consensus problem to include both traditional consensus and this weaker version. A straightforward generalization of the Paxos consensus algorithm implements general consensus. The generalizations of consensus and of the Paxos algorithm require a mathematical *detour de force* into a type of object called a command-structure set.

Engineer's Abstract

The state-machine approach to implementing a fault-tolerant distributed system involves reaching agreement on the sequence of system commands by executing a sequence of separate instances of a consensus algorithm. It can be shown that any fault-tolerant asynchronous consensus algorithm requires at least two message delays between the issuing of a command and when it can be executed. But even in the absence of faults, no algorithm can guarantee this fast an execution if two different commands are issued concurrently. We generalize the state-machine approach to involve reaching agreement on a partially ordered set of commands. By generalizing the Paxos consensus algorithm, we can implement a system in which concurrently issued commands can always be executed in two message delays if they are non-interfering, so it does not matter in which order those commands are executed. For many systems, concurrent commands are rarely interfering, so the generalized Paxos algorithm can be quite efficient. And command-structure sets are very simple.

Contents

1 Introduction

A system in which clients issue commands and receive responses can be represented as a state machine. Executing a command in a given state produces an output and a new state. A simple example is a toy banking system in which a client can deposit or withdraw money. The state consists of the amount of money in each client's account, and clients can issue *deposit* or *withdraw* commands. Executing the command c *withdraws \$100* in a state with at least \$100 in client c's account subtracts \$100 from the amount in that account and produces as output *\$100 for* c, which is some string of bits that conveys \$100 in digital cash to c. Executing c *deposits \$50* adds \$50 to c's account and produces OK as output.

In the standard state-machine approach, a sequence of instances of a consensus algorithm are used to choose the sequence of client commands. The i^{th} instance of the algorithm chooses the i^{th} command to be executed. Given an initial state, the sequence of commands defines the output and new state produced by executing each command in the sequence.

Classic Paxos [4] provides a fault-tolerant implementation of an arbitrary state machine in an asynchronous message-passing system. In Paxos, clients send commands to a leader. During normal operation, the leader receives a client's command, assigns it a new command number i, and then begins the i^{th} instance of the consensus algorithm by sending what are called *phase 2a* messages to a set of *acceptor* processes. We ignore for now what the acceptors do when they receive those messages.

A previously unpublished algorithm, called Fast Paxos, tries to save one message delay by having the client send its command directly to the acceptors, bypassing the leader. An acceptor interprets the client's message as if it were a phase 2a message from the leader for the next unused command number—that is, the command number that the acceptor believes to be the next unused one.

Fast Paxos works fine if all acceptors assign the same command number to a client's command. However, suppose all acceptors think the next unused command number is 42, at which point client c_A issues command A and client c_B concurrently issues command B. Some acceptors may get c_A's message first, assigning command number 42 to A, and then receive c_B's message, assigning 43 to B. Others may receive the messages in the opposite order, making the opposite assignment of command numbers to commands. This can cause instances 42 and 43 of the consensus algorithm not to choose any command right away, forcing the leader to intercede. Resolving this collision adds at least one more message delay.

In Section 2.3, we sketch a theorem asserting that deciding whether A or B is the next command has to add an extra message delay to Fast Paxos. If it matters in which order commands A and B are executed—for example, if A deposits money in an account and B reads the account's balance—then there is no way to avoid this extra message delay. However, suppose A and B *commute*, meaning that executing them in either order has the same effect—for example, if they are operations to two different accounts. There is then no inherent need to decide which to execute first. This suggests that we should be able to get the speed of Fast Paxos even when two commands are issued concurrently, if those two commands commute. In many systems, concurrently issued commands almost always commute. An implementation that saves a message delay in almost all cases can be significantly more efficient than one using the conventional state-machine approach.

By choosing an execution sequence, the state-machine approach decides the order in which all pairs of commands are executed—even pairs of commuting commands. Instead of choosing a sequence, it suffices to choose a partially ordered set of commands in which any two non-commuting commands are ordered. We call such a partially ordered set a *command history.* Executing the commands in a command history in any order consistent with its partial order has the same effect. We generalize the state-machine approach by choosing a history rather than a sequence of commands.

The key to this generalization lies in generalizing the concept of consensus. The customary view of the state-machine approach is that it solves a sequence of separate consensus problems to choose a sequence of commands, where consensus means reaching agreement on a single value. Instead, we think of it as solving a single more general consensus problem that requires agreeing on an increasing set of values—namely, the currently chosen prefix of the command sequence. We encompass traditional consensus, agreement on command sequences, and agreement on histories by a general consensus problem of reaching agreement on an increasing set of values in a partially ordered set with an *append* operation—a type of set we call a *command structure set.* The Paxos consensus algorithm can be directly extended to solve the generalized consensus problem for values in an arbitrary command structure set.

Section 2 motivates our search for faster algorithms by describing some lower bounds for traditional consensus. Section 3 generalizes the traditional consensus problem, which chooses a single value, to the problem of choosing monotonically increasing, consistent values. Section 4 is a mathematical detour in which we define command-structure sets and give some examples. We then present the generalized Paxos algorithm in Section 5 and briefly discuss its implementation in Section 6. A concluding section summarizes what we have done. Proofs and proof sketches are relegated to the appendix. Section C of the appendix also contains TLA$^+$ specifications of command structure sets and of our algorithms.

Generalized consensus for command histories is equivalent to the concept of generic consensus introduced by Pedone and Schiper [9]. Generalized Paxos is quite similar to their algorithm $\mathcal{GB}+$ for this set of c-structs, but theirr algorithm can detect conflicts and incur the expense of resolving them in cases when generalized Paxos does not. Moreover, generalized Paxos is a natural extension of Fast Paxos and shares its flexibility—in particular, it can switch to ordinary Paxos when too many failures have occurred to permit fast execution. Pedone and Schiper's algorithm employs an *ad hoc* prefix to an ordinary consensus algorithm, and they did not consider switching from fast to slower execution.

2 Traditional Consensus

We now review the conventional consensus problem and give some lower bounds on solutions. We restrict ourselves to consensus in asynchronous systems.

2.1 The Requirements

The consensus problem is typically described in terms of agreement among a single set of processes. However, it is better to cast the problem in terms of a set of *proposer* processes that propose values and a set of *learner* processes that must agree upon a value. Think of the proposers as a system's clients and the learners as the servers that cooperate to implement the system.

The traditional consensus problem has three safety requirements:

Nontriviality Any value learned must have been proposed.

Stability A learner can learn at most one value. (In other words, it cannot change its mind about what value it has learned.)

Consistency Two different learners cannot learn different values.

(Stability is often only tacitly assumed.) The safety properties are required to hold under certain failure assumptions. For asynchronous systems, they are generally required to hold despite any number of non-Byzantine failures.

The traditional consensus problem also has the following liveness requirement:

Liveness(C, l) If value C has been proposed, then eventually learner l will learn some value.

This requirement is stated in terms of C and l because the assumption under which it must hold generally depends on these parameters. For asynchronous implementations, the usual assumption is that learner l, the proposer of C, and a sufficient number of other processes are nonfaulty and can communicate with one another.

2.2 Acceptors and Quorums

Consensus is implemented using a finite set of processes called *acceptors*. Acceptors, proposers, and learners are processes. Each process is executed on a node, and a single node may execute several of these processes. A set of nodes is considered to be nonfaulty iff the set of processes executed by those nodes is nonfaulty.

A *quorum* Q is a set of acceptors that is large enough to ensure liveness. More precisely, Q is a quorum iff condition Liveness(C, l) of consensus holds when the set containing the process that proposed command C, the learner l, and the acceptors in Q is nonfaulty for a long enough period of time. (How long is "long enough" depends on the algorithm.) This condition is required to hold regardless of what may have happened before this set of processes became nonfaulty.

Proposer, acceptor, and learner can be viewed as roles performed by the nodes in a distributed system. The relation between these roles and the roles of client and server can vary from system to system. Typically, the clients are proposers and the server nodes are both learners and acceptors. There may be additional nodes that act only as acceptors—nodes sometimes called *witnesses*. The clients may also be considered to be learners. What roles a node plays is an implementation choice that determines the fault-tolerance properties of the system.

2.3 Lower-Bound Results for Consensus

When devising an algorithm, it helps to know what is possible. So before deriving the generalized Paxos consensus algorithm, we describe some lower-bound results for consensus in asynchronous systems.

The precise statements of the lower-bound results are tricky. They require some unobvious hypotheses, and algorithms that violate the bounds are possible in certain special cases. One obvious hypothesis that we state once and for all now is that there are at least two proposers and two learners.

(Consensus becomes a trivial problem if there is only one acceptor or one learner.) We omit the less obvious hypotheses and state only approximate versions of the results. The ideas behind their proofs are given in Section A of the appendix. The precise statements of the results and their rigorous proofs appear elsewhere [7].

The first result is rather obvious; equivalent results have appeared before.

Approximate Theorem 1 *Any two quorums have non-empty intersection.*

A consensus algorithm using N acceptors is said to tolerate F faults if every set of $N - F$ acceptors is a quorum. Approximate Theorem 1 implies that this is possible only if $N > 2F$.

The other results give lower bounds on the number of message delays required to learn a value. The message delay between two events in the execution of an algorithm is the length of the longest message chain connecting the two events. A message chain is a sequence of messages, each of which is received by the sender of the next message before that next message is sent. A message chain connects event e to event f if its first message is sent by the process executing e when or after it executes e, and whose last message is received by the process executing f before or when it executes f.

We say that a learner l learns a value in k message delays in an execution in which l learns a value C, and there are k message delays between the event of proposing C and the event of l learning C. The following result is also fairly obvious and has been proved in several settings [1].

Approximate Theorem 2 *Learning is impossible in fewer than 2 message delays.*

We define a set Q of acceptors to be a *fast quorum* iff, for every proposer p and learner l, there is an execution involving only the set of processes $Q \cup \{p, l\}$ in which a value proposed by p is learned by l in two message delays. Events that do not influence l's learning event are irrelevant, and processes could fail before they are performed. We can therefore assume that the communication in such an execution consists only of p proposing a value and sending messages to the acceptors in Q and to l, followed by the acceptors in Q sending messages to l.

Comparing the following result with Approximate Theorem 1 shows that fast quorums have to be bigger than plain quorums. We believe that this result was first announced in [6] and first proved in [7].

Approximate Theorem 3 *If Q_1 and Q_2 are fast quorums and Q is a quorum, then $Q_1 \cap Q_2 \cap Q$ is non-empty.*

We say that an algorithm with N acceptors is fast learning despite E faults iff every set of $N - E$ acceptors is a fast quorum. Approximate Theorem 3 implies that such an algorithm that tolerates F faults is possible only if $N > 2E + F$.

The following result shows that it is impossible for a fault-tolerant consensus algorithm to guarantee, even in the absence of failure, that a value is always learned in two message delays. Its proof shows that this can't occur because two proposers can concurrently propose different values. The result is an approximate special case of the Collision-Fast Learning theorem of [7].

Approximate Theorem 4 *If, for every acceptor a, there is a quorum not containing a, then a consensus algorithm cannot ensure that, even in the absence of failures, every learner learns a value in two message delays.*

3 Generalized Consensus

We now generalize the consensus problem from agreeing on a single value to agreeing on an increasing set of values. We start with the problem of agreeing on a growing sequence of commands—the problem that must be solved in the conventional state-machine approach. Let $learned[l]$ be learner l's current knowledge of the command sequence. The value of $learned[l]$ will change over time as l learns more commands in the sequence.

In the conventional state-machine approach, a learner need not learn the sequence of commands in order. It might learn the 5^{th} command in the sequence before learning the 3^{rd} command. (However, it knows that the command is the 5^{th} one.) It is convenient to define learning so that a learner is not considered to have learned the 5^{th} command until it has learned the preceding four commands. This is a reasonable notion of learning, since a server cannot execute the 5^{th} command until it knows the first four. We therefore let $learned[l]$ always be a sequence of commands (with no gaps), for every learner l.

The four requirements for traditional consensus can be generalized as follows. The prefix relation on sequences is the usual reflexive one, in which any sequence is a prefix of itself.

Nontriviality For any learner l, the value of $learned[l]$ is always a sequence of proposed commands,

Stability For any learner l, the value of $learned[l]$ at any time is a prefix of its value at any later time.

Consistency For any learners l_1 and l_2, it is always the case that one of the sequences $\mathit{learned}[l_1]$ and $\mathit{learned}[l_2]$ is a prefix of the other.

Liveness(C, l) If command C has been proposed, then eventually the sequence $\mathit{learned}[l]$ will contain the element C.

We now abstract these four consensus requirements from sequences to a more general set of values that we call *command structures*, or *c-structs* for short. C-structs are formed from a "null" element, which we call $\bot$, by the operation of appending commands. More precisely, a c-struct set is a set, containing the element $\bot$, with an append operator $\bullet$ such that $v \bullet C$ is a c-struct, for any c-struct v and command C. We extend $\bullet$ to sequences of commands in the usual way, defining $v \bullet \langle C_1, \ldots, C_m \rangle$ to equal $v \bullet C_1 \bullet \cdots \bullet C_m$. C-structs are more general than command sequences because $\bot \bullet \sigma$ can equal $\bot \bullet \tau$ for two different command sequences σ and τ.

We now generalize the requirements for consensus from sequences to c-structs. The generalizations of nontriviality, stability, and liveness are fairly obvious.

Nontriviality For any learner l, there always exists a sequence σ of proposed commands such that $\mathit{learned}[l] = \bot \bullet \sigma$.

Stability For any learner l, if the value of $\mathit{learned}[l]$ at any time is v, then at all later times there exists a command sequence σ such that $\mathit{learned}[l] = v \bullet \sigma$.

Liveness(C, l) If command C has been proposed, then $\mathit{learned}[l]$ eventually equals $v \bullet C \bullet \sigma$, for some c-struct v and command sequence σ.

The obvious generalization of consistency is to use the same condition as before, where v is defined to be a prefix of w iff $w = v \bullet \sigma$ for some command sequence σ. However, it turns out that we want to generalize still further by requiring only that any two learned c-structs are both prefixes of the same c-struct. (This is obviously true if one is a prefix of the other.) So, our generalized condition is:

Consistency For all learners l_1 and l_2, there always exist command sequences σ_1 and σ_2 such that $\mathit{learned}[l_1] \bullet \sigma_1 = \mathit{learned}[l_2] \bullet \sigma_2$.

Section C.2 of the appendix contains a TLA$^+$ specification of generalized consensus.

4 Command-Structure Sets

In Section 3, we informally introduced the concept of a c-struct to define the generalized consensus problem. We now formalize the notion of a set of c-structs and give some examples of such sets. But first we review some standard notation and simple mathematics.

4.1 Mathematical Preliminaries

4.1.1 Notation

Whenever possible, we use an informal style to describe mathematical concepts. But mathematics can sometimes be quite hard to understand when written in prose. While we try to keep the mathematics as prosaic as possible, there are times when words are inadequate and formulas are required. We introduce here some fairly standard mathematical notation. We use the customary operators of propositional logic and predicate calculus, with $\forall x \in S : P$ and $\exists x \in S : P$ asserting that P holds for all and for some x in S, respectively, and we let $\triangleq$ mean *is defined to equal*. We use the following notation for representing sets.

- $\{e_1, \ldots, e_n\}$ is the set consisting only of the elements e_i. In particular, $\{\,\}$ is the empty set.
- $\{x \mid P\}$ is the set of all x such that P is true.
- $\{x \in S \mid P\}$ equals $\{x \mid (x \in S) \wedge P\}$.

For example, we define the union ($\bigcup$) and intersection ($\bigcap$) of a set $\mathcal{S}$ of sets as follows:

- $\bigcup \mathcal{S} \;\triangleq\; \{x \mid \exists T \in \mathcal{S} : x \in T\}$
- $\bigcap \mathcal{S} \;\triangleq\; \{x \mid \forall T \in \mathcal{S} : x \in T\}$

4.1.2 Sequences

We use the term *sequence* to mean a finite sequence. We enclose sequences in angle brackets, so $\langle e_1, \ldots, e_m \rangle$ is a sequence with elements e_i, and $\langle\,\rangle$ is the empty sequence. We let $Seq(S)$ be the set of all sequences whose elements are in the set S, and let $\circ$ be the usual concatenation operator:

$$\langle e_1, \ldots, e_m \rangle \circ \langle f_1, \ldots, f_n \rangle \;\triangleq\; \langle e_1, \ldots, e_m, f_1, \ldots, f_n \rangle$$

For conciseness, we use the term *c-seq* for a finite sequence of commands.

4.1.3 Partial Orders

A relation $\sqsubseteq$ is a (reflexive) partial order on a set S iff it satisfies the following properties, for all u, v, and w in S:

- $u \sqsubseteq v$ and $v \sqsubseteq u$ iff $u = v$.
- $u \sqsubseteq v$ and $v \sqsubseteq w$ imply $u \sqsubseteq w$.

For a partial order $\sqsubseteq$, we define $v \sqsubset w$ to mean $v \sqsubseteq w$ and $v \neq w$.

Given a partial order $\sqsubseteq$ on S, a lower bound of a subset T of S is an element v of S such that $v \sqsubseteq w$ 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 of T. If a glb exists, then it is unique. We write the glb of T as $\sqcap T$, and we let $v \sqcap w$ equal $\sqcap \{v, w\}$ for any v and w in S.

Upper bounds and the least upper bound (lub) of a set T are defined in the analogous fashion. We write the lub of T as $\sqcup T$ (remember "l$\sqcup$b"), and we let $v \sqcup w$ equal $\sqcup \{v, w\}$ for any v and w in S.

The TLA$^+$ module *OrderRelations* in Section C.1 of the appendix formalizes these definitions.

4.1.4 Equivalence Classes

A relation $\sim$ on a set S is called an *equivalence relation* iff it satisfies the following properties, for all u, v, and w in S:

- $u \sim u$
- $u \sim v$ iff $v \sim u$
- $u \sim v$ and $v \sim w$ imply $u \sim w$.

We define $[u]$, the equivalence class under $\sim$ of an element u in S, by

$$[u] \;\triangleq\; \{v \in S \mid v \sim u\}$$

Thus, $u \sim v$ iff $[u] = [v]$. The set of all such equivalence classes is called the *quotient space of S under $\sim$*, and is written $S/\sim$.

4.1.5 Directed Graph

A *directed graph* consists of a pair $\langle N, E \rangle$ where N (the set of nodes) is a set and E (the set of edges) is a subset of $N \times N$ (the set of pairs of nodes). A *subgraph* of a directed graph $\langle N, E \rangle$ is a directed graph $\langle M, D \rangle$ where $M \subseteq N$ and $D = E \cap (M \times M)$. The subgraph $\langle M, D \rangle$ is defined to be a *prefix* of $\langle N, E \rangle$ iff for every edge $\langle m, n \rangle$ in E, if n is in M then m is in M.

4.2 C-Struct Sets

Let *Cmd* be the set of all commands and *CStruct* be the set of all c-structs. So far, all we have assumed about the set *CStruct* is that it contains an element $\perp$ and an operator $\bullet$ that appends a command to a c-struct. We now introduce four axioms CS1–CS4 on *CStruct*. In other words, a set *CStruct* containing an element $\perp$ and an append operator $\bullet$ is defined to be a c-struct set iff these axioms are satisfied. The definitions of this section are formalized in TLA$^+$ module *CStructs* of Appendix Section C.1.

We inductively define $v \bullet \langle C_1, \ldots, C_m \rangle$ for a c-struct v and a c-seq $\langle C_1, \ldots, C_m \rangle$ by:

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

Our first axiom asserts that every c-struct is obtained by concatenating a c-seq to $\perp$.

CS1. $\mathit{CStruct} = \{\perp \bullet \sigma \mid \sigma \in \mathit{Seq}(\mathit{Cmd})\}$

It follows from CS1 that *CStruct* is isomorphic to the quotient space of $\mathit{Seq}(\mathit{Cmd})$ under the equivalence relation defined by $\sigma \sim \tau$ iff $\perp \bullet \sigma = \perp \bullet \tau$.

We define the prefix relation $\sqsubseteq$ on c-structs by

$v \sqsubseteq w$ iff there exists a c-seq σ such that $w = v \bullet \sigma$.

for any c-structs v and w. The next assumption we make about c-structs is:

CS2. $\sqsubseteq$ is a partial order on the set of c-structs.

It is not hard to show that CS2 is equivalent to the assumption

$v \bullet \sigma \bullet \tau = v$ implies $v \bullet \sigma = v$, for every c-struct v and command sequences σ and τ.

As is customary with a partial order, we read $\sqsubseteq$ as "less than or equal". But because $\sqsubseteq$ is a prefix relation, we also read $v \sqsubseteq w$ as "v is a prefix of w" or "w is an extension of v".

We say that the c-struct $\perp \bullet \langle C_1, \ldots, C_m \rangle$ is *constructible* from the commands C_i. We define $\mathit{Str}(P)$ to be the set of all c-structs that can be constructed from elements of the set P of commands:

$$\mathit{Str}(P) \;\triangleq\; \{\perp \bullet \sigma \mid \sigma \in \mathit{Seq}(P)\}$$

CS1 asserts that *CStruct* equals $\mathit{Str}(\mathit{Cmd})$.

We define two c-structs v and w to be *compatible* iff they have a common upper bound—that is, iff there is some c-struct z with $v \sqsubseteq z$ and $w \sqsubseteq z$. A set S of c-structs is called *compatible* iff every pair of elements in S are compatible.

The generalized Paxos algorithm computes lubs and glbs of c-structs. It requires that any non-empty finite set S of c-structs has a glb, and if S is compatible, then it has a lub. Moreover, it requires that its glb and lub be constructible from the same commands as the elements of S. In other words:

> **CS3.** For any set P of commands 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.

It follows from CS3 that for any finite set S of c-structs in $Str(P)$, if S is non-empty then $\sqcap S \in Str(P)$, and if S is compatible then $\sqcup S \in Str(P)$. (The definition of lub implies that $\perp$ is the lub of the empty set.) Letting P equal the set Cmd, CS3 asserts the existence of the glb of any non-empty finite set S of c-structs, and the existence of its lub if S is compatible.

For any compatible set S of c-structs, we define the $\sqcup$*-completion* of S to be the set of all lubs of finite subsets of S—that is, the set

$$\{\sqcup T \mid (T \subseteq S) \wedge (T \text{ finite})\}$$

The set S is a subset of its $\sqcup$-completion, since $\sqcup \{v\} = v$, for any c-struct v. The $\sqcup$-completion of any compatible set is compatible, since $(\sqcup T) \sqcup (\sqcup U) = \sqcup (T \cup U)$ for any compatible finite sets T and U.

We say that a c-struct v *contains* a command C iff v is constructible from some set of commands containing C. Equivalently, v contains C iff $v = \perp \bullet \sigma \bullet C \bullet \tau$ for some c-seqs σ and τ. A c-struct v can contain a command C even if v is constructible from commands not containing C. That is, v could contain C and be an element of $Str(P)$ for some set P of commands that does not contain C. This is because we could have $\perp \bullet \sigma = \perp \bullet \tau$ even though the c-seqs σ and τ have no commands in common. For example, commands could have some irrelevant field that is thrown away by the append operator $\bullet$. We would then have $\perp \bullet \sigma = \perp \bullet \tau$ for any τ obtained from σ by changing just that field in all the elements of σ.

It follows easily from the definition of $\sqsubseteq$ that if $v \sqsubseteq w$ and v contains C then w also contains C. Hence, the lub of any non-empty set of compatible c-structs that contain C also contains C. We need to assume this for glbs as well.

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

It follows from CS4 that if all the elements of a finite, non-empty compatible set S of c-structs contain C, then $\sqcap S$ also contains C.

The operator $\bullet$ combines c-structs and c-seqs. Mathematicians like more symmetric operators, so it is natural to try to define $v \bullet w$ when v and w are both c-structs. The obvious definition is:

$$v \bullet (\perp \bullet \sigma) \triangleq v \bullet \sigma, \text{ for any c-struct } v \text{ and c-seq } \sigma.$$

For this to uniquely define $\bullet$ on c-structs, we need the addition assumption:

> **Monoid Assumption** For all c-structs v and c-seqs σ and τ, if $\perp \bullet \sigma = \perp \bullet \tau$ then $v \bullet \sigma = v \bullet \tau$.

We can then extend the operator $\bullet$ to be a "multiplication" operator on *CStruct*. The set *CStruct* with this operator is called a *monoid* with identity element $\perp$. We say that a c-struct set is *monoidal* iff it satisfies the monoid assumption. The generalized Paxos algorithm does not require *CStruct* to be monoidal. However, certain optimizations are possible if it is.

One might expect that there should not exist an infinite descending chain $v_1 \sqsupset v_2 \sqsupset v_3 \sqsupset \cdots$ of c-structs, each an extension of the next. We leave to the reader the exercise of finding a c-struct set that allows such infinite descending chains. That the generalized Paxos algorithm works even on such a c-struct set seems to be of no practical significance.

4.3 The Consensus Requirements Restated

We now restate the requirements for consensus in terms of the concepts defined above, where *propCmd* is the set of all proposed commands:

Nontriviality $learned[l] \in Str(propCmd)$ always holds, for every learner l.

Stability It is always the case that $learned[l] = v$ implies $v \sqsubseteq learned[l]$ at all later times, for any learner l and c-struct v.

Consistency $learned[l_1]$ and $learned[l_2]$ are always compatible, for all learners l_1 and l_2.

Liveness(C, l) If $C \in propCmd$ then eventually $learned[l]$ contains C.

The approximate theorems of Section 2.3 above also hold (approximately) for the generalized consensus problem if there exist commands C and D such that $\perp \bullet C$ and $\perp \bullet D$ are not compatible.

4.4 Some Examples of C-Structs

The most obvious example of a c-struct set is the set $Seq(Cmd)$ of all c-seqs, where $\bullet$ is the usual append operator and $\sqsubseteq$ the ordinary prefix relation. Two c-seqs are compatible iff one is a prefix of the other. It is easy to see that the set of c-seqs satisfies C1–C4, so it is indeed a c-struct set, and that it is monoidal.

As we have seen, the generalization of consensus to this c-struct set is essentially the problem solved by the conventional state-machine approach. We now introduce some other c-struct sets that lead to different consensus problems.

Nonduplicate Command Sequences

The traditional state-machine approach, specified by letting *CStruct* equal *Seq*(*Cmd*), allows multiple copies of a proposed command to appear in the command sequence. A banking system should not allow a single *c withdraws \$100* command to withdraw more than \$100. The problem of multiple executions of the same command is traditionally solved by making commands idempotent. The state machine used to implement the banking system is defined so that only the first execution of any single withdrawal or deposit command has any effect. A command contains a unique identifier (uid), so c can withdraw \$200 by issuing two different *c withdraws \$100* commands. (The uid also appears on the command's output, so the client knows to which command a response is for.)

An alternative way of handling the problem of duplicate commands is to solve a different consensus problem that eliminates them. We do this by taking *CStruct* to be the set of c-seqs with no duplicates—that is, c-seqs $\langle C_1, \ldots, C_m \rangle$ such that the C_i are all distinct. For any command C and sequence $\langle C_1, \ldots, C_m \rangle$ without duplicates, we define

$$\langle C_1, \ldots, C_m \rangle \bullet C \;\triangleq\; \begin{cases} \langle C_1, \ldots, C_m \rangle & \text{if } C \text{ equals some } C_i, \\ \langle C_1, \ldots, C_m, C \rangle & \text{otherwise} \end{cases}$$

As with ordinary sequences, $\sqsubseteq$ is the prefix relation and two c-structs are compatible iff one is a prefix of the other. It is easy to see that this is a c-struct set (satisfies axioms CS1–CS4) and that it is monoidal.

Redefining consensus in this way does not make it any easier to solve the problem of multiple executions of the same command. It just transfers the problem from the state machine to the computation of $\bullet$. Instead of duplicate commands being detected when executing the state machine, they are detected when executing the consensus algorithm.

Commands with $\perp$

We now show that generalized consensus generalizes the ordinary consensus problem of choosing a single command. To obtain the ordinary consensus problem, we define *CStruct* be the set $Cmd \cup \{\perp\}$ consisting of all commands together with the additional element $\perp$. We define $\bullet$ by

$$v \bullet C \triangleq \begin{cases} C & \text{if } v = \perp, \\ v & \text{otherwise} \end{cases}$$

This implies that $v \sqsubseteq w$ iff $v = \perp$ or $v = w$. Two commands are compatible iff they are equal, and every command contains every command, since $C = C \bullet \langle D \rangle$ for any commands C and D. It is not hard to check that the set $Cmd \cup \{\perp\}$ and the operation $\bullet$ form a monoidal c-struct set.

With this c-struct set, generalized consensus reduces to the traditional consensus problem, where $learned[l] = \perp$ means that l has not yet learned a value. For example, the consistency condition of generalized consensus asserts that, for any two learners l_1 and l_2, either $learner[l_1]$ and $learner[l_2]$ are equal, or one of them equals $\perp$.

Command Sets

A very simple consensus problem is obtained by taking *CSruct* to be the set of all finite sets of commands. We let $\perp$ be the empty set and define $v \bullet C$ to equal $v \cup \{C\}$.

This c-struct set is interesting because the resulting consensus problem is very easy to solve. Proposers simply send commands to learners, and a learner l adds a command to $learned[l]$ whenever it receives a proposer's message containing that command.

We can also let *CStruct* be the set of all finite multisets of commands, where a multiset is a set in which an element can appear multiple times. The definition of $\perp$ and $\bullet$ are the same, except where $\cup$ is taken to be multiset union.

It is not too hard to see that these two different choices of *CStruct* yield equivalent consensus problems. In particular, an algorithm in which a learner l keeps only a single copy of any command in $learned[l]$ also solves the consensus problem for multisets of commands. This observation illustrates that our statement of the consensus problem considers multiple proposals of the same command to be the same as a single proposal.

Command Histories

As observed in the introduction, defining an execution of a set of commands does not require totally ordering them. It suffices to determine the order in which every pair of non-commuting commands are executed. Determining whether or not two commands commute can be difficult. Instead, one generally introduces an *interference* relation $\asymp$ (also called a *dependence* relation) and requires that $C \asymp D$ holds for any non-commuting pair C, D of commands. We can allow $C \asymp D$ to hold for some pairs C, D of commuting commands as well. For the banking example, we can define $C \asymp D$ to be true iff commands C and D both access the same account. Then $C \asymp D$ holds even if C and D just deposit money into the same account, so they commute.

In general, we assume a symmetric relation $\asymp$ on commands—that is a relation satisfying $C \asymp D$ iff $D \asymp C$ for any commands C and D. We define the equivalence relation $\sim$ on c-seqs by letting two sequences be equivalent iff one can be transformed into the other by permuting elements in such a way that the order of all pairs of interfering commands is preserved. The precise definition is:

> $\langle C_1, \ldots, C_m \rangle \sim \langle D_1, \ldots, D_n \rangle$ iff $m = n$ and there exists a permutation π of $\{1, \ldots, m\}$ such that, for each $i, j = 1, \ldots, m$:
>
> - $D_i = C_{\pi(i)}$
> - If $i < j$ and $C_i \asymp C_j$ then $\pi(i) < \pi(j)$.

We define a *command history* to be an equivalence class of c-seqs under this equivalence relation. Command histories are isomorphic to Mazurkiewicz traces [8], which were introduced to study the semantics of concurrent systems. For conciseness, we usually write *history* instead of *command history.*

We now let *CStruct* be the set of all histories (that is, the quotient space $Seq(Cmd)/\sim$), and we define $\bullet$ by

$$[\langle C_1, \ldots, C_m \rangle] \bullet C \;\triangleq\; [\langle C_1, \ldots, C_m, C \rangle]$$

for any c-seq $\langle C_1, \ldots, C_m \rangle$ and command C. It is easy to check that, for any two c-seqs σ and τ, if $[\sigma] = [\tau]$ then $[\sigma \circ \langle C \rangle] = [\tau \circ \langle C \rangle]$ for any command C. Therefore, this uniquely defines the operator $\bullet$.

To show that the set of histories is a c-struct, we observe that the history $[\langle C_1, \ldots, C_m \rangle]$ is isomorphic to a directed graph $G(\langle C_1, \ldots, C_m \rangle)$ whose nodes are the C_i, where there is an edge from C_i to C_j iff $i < j$ and $C_i \asymp C_j$. To define the mapping G from c-seqs to graphs precisely, we must distinguish different occurrences of the same command in a history. So we define the nodes of $G(\langle C_1, \ldots, C_m \rangle)$ to consist of all pairs $\langle C_i, k_i \rangle$ where C_i is the ${k_i}^{\text{th}}$ occurrence of the command C_i in the sequence $\langle C_1, \ldots, C_m \rangle$. There is an edge from $\langle C_i, k_i \rangle$ to $\langle C_j, k_j \rangle$ iff $i < j$ and $C_i \asymp C_j$. For example, suppose that C, D, and E are distinct commands with $C \asymp D$, $C \asymp E$, $D \not\asymp E$, and no command interferes with itself. Then $G(\langle C, D, E, C, E \rangle)$ is the graph

$$
\begin{tikzcd}
 & \langle D,\, 1\rangle \arrow[dr] & & \\
\langle C,\, 1\rangle \arrow[ur] \arrow[dr] \arrow[rrr, bend left] & & \langle C,\, 2\rangle \arrow[r] & \langle E,\, 2\rangle \\
 & \langle E,\, 1\rangle \arrow[ur] & &
\end{tikzcd}
$$

It is not hard to see that for any two c-seqs σ and τ:

- $[\sigma] = [\tau]$ iff $G(\sigma) = G(\tau)$.
- $[\sigma] \sqsubseteq [\tau]$ iff $G(\sigma)$ is a prefix of $G(\tau)$.
- $[\sigma]$ and $[\tau]$ are compatible iff the subgraphs of $G(\sigma)$ and $G(\tau)$ consisting of the nodes they have in common are identical, and $C \not\asymp D$ for every node $\langle C,\, i \rangle$ in $G(\sigma)$ that is not in $G(\tau)$ and every node $\langle D,\, j \rangle$ in $G(\tau)$ that is not in $G(\sigma)$.

Using these observations, one can show that the set of histories is a monoidal c-struct set.

If non-interfering commands commute, then it is easy to see that executions of equivalent c-seqs yield the same result—that is, the same output for each command and the same final state. Hence, to use the state-machine approach for implementing a system, it suffices to solve the consensus problem for histories.

If $\asymp$ is defined so that all pairs of commands are interfering, then histories are equivalent to sequences. If $\asymp$ is defined so no commands interfere, then histories are equivalent to finite multisets of commands. Intuitively, the weaker the $\asymp$ relation is (the less likely it is for commands to interfere), the easier it is to solve the generalized consensus problem. As we have seen, in the limiting case of command sets, the problem has a trivial solution.

Our definition of a history allows it to contain duplicate commands. As we did with sequences, we can also define c-structs to consist of histories without duplicates. (A history without duplicates is an equivalence class of sequences without duplicates.) As with sequences, this moves the problem of duplicate detection from the state machine to the consensus algorithm.

5 The Generalized Paxos Consensus Algorithm

We now develop an algorithm to implement generalized consensus in an asynchronous, non-Byzantine distributed setting. This means that we assume a network of processes that communicate by sending messages. Messages can be lost, but they cannot be corrupted. A process can fail by stopping and doing nothing, but it cannot execute its algorithm incorrectly. We make no assumption about relative execution speeds or about the time it takes for a message to be delivered.

We require the safety conditions for consensus to hold with no additional assumptions. The famous theorem of Fischer, Lynch, and Paterson [3] implies that additional assumptions are needed to ensure liveness. We defer a discussion of liveness to Section 6.2. (However, the need to satisfy liveness will motivate our development of the algorithm.)

Section C of the appendix contains formal TLA$^+$ specifications of our algorithms. The definitions of Sections 5.1 and 5.2 are formalized in Section C.3. Appendix Section C.4 contains a TLA$^+$ specification of the abstract algorithm of Section 5.3. The distributed abstract algorithm of Section 5.4 and the generalized Paxos consensus algorithm of Section 5.5 are formalized in the TLA$^+$ specifications of Sections C.5 and C.6, respectively.

5.1 Ballots and Quorums

Like ordinary Paxos, the generalized Paxos algorithm executes a sequence of numbered *ballots* to choose values. If a ballot does not succeed in choosing values because of a failure, then a higher-numbered ballot is executed. Each acceptor participates in only one ballot at a time, going from one ballot only to a higher-numbered one. However, different acceptors may be participating in different ballots at the same time.

We assume an unbounded set of ballot numbers that are totally ordered by a relation $<$, with a smallest ballot number that we call 0. The natural numbers is an obvious choice for the set of ballot numbers, but not the only one. The set of all non-negative reals is another possible choice. We write *balnum* as an abbreviation for *ballot number*, and we let *BalNum* be the set of all balnums.

We assume that some ballot numbers are designated to be *fast*. Ballots with fast balnums will be used to try to have learners learn commands in two message delays. A balnum that is not fast is said to be a *classic* balnum. We assume that the sets of fast and classic balnums are both unbounded.

A quorum of acceptors must participate in a ballot for that ballot to succeed in choosing values. What constitutes a quorum can depend upon the ballot number. For each balnum m, we assume a collection of sets of acceptors called m-quorums. We let $Quorum(m)$ be the set of all m-quorums. Approximate Theorems 1 and 3 lead us to make the following assumption about quorums:

Quorum Assumption For all balnums k and m:

1. The intersection of any k-quorum and any m-quorum is non-empty.
2. If k is a fast balnum, then the intersection of any two k-quorums and any m-quorum is non-empty. More precisely:

$$\begin{array}{l} \forall\, Q_1,\ Q_2 \in Quorum(k),\ R \in Quorum(m) \ : \\ \quad Q_1 \cap Q_2 \cap R \neq \{\} \end{array}$$

5.2 Ballot Arrays

The state of the balloting is described by a data structure called a *ballot array*. Before talking about algorithms, we define ballot arrays and prove some properties about them.

We first introduce a new value *none* that is not a c-struct, and we extend $\sqsubseteq$ to c-structs and *none* by defining $none \sqsubseteq none$ to be true and $v \sqsubseteq w$ to be false if either v or w (but not both) equals *none*. We define a ballot array as follows.

Definition 1 A *ballot array* β is a mapping that assigns to each acceptor a a balnum $\widehat{\beta}_a$ and to each acceptor a and balnum m a value $\beta_a[m]$ that is a c-struct or equals *none*, such that for every acceptor a:

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

Think of a ballot array β as describing a possible state of a voting algorithm. The value of $\widehat{\beta}_a$ represents the number of the ballot in which a is currently participating. The value of $\beta_a[m]$ represents the votes cast by a in the ballot numbered m; if $\beta_a[m] = none$ then a has not voted in ballot m, otherwise a has voted for all prefixes of $\beta_a[m]$. But this is only how we think about ballot arrays. A ballot array is really just any data structure satisfying the definition.

We now define a c-struct to be *chosen* in a ballot array iff it is the glb of the c-structs voted for in some ballot by a quorum:

Definition 2 A c-struct v is *chosen at* balnum m *in* ballot array β iff there exists an m-quorum Q such that $v \sqsubseteq \beta_a[m]$ for all acceptors a in Q. A c-struct v is *chosen in* ballot array β iff it is chosen at m in β for some balnum m.

Remember that if v is a c-struct, then $v \sqsubseteq w$ implies $w \neq none$. Hence, for a c-struct to be chosen at m, there must be at least one m-quorum Q all of whose members have voted in ballot m.

We next define what it means for v to be *choosable* at balnum m in β. Intuitively, it means that β represents a state of a voting algorithm in which it is possible for acceptors to cast additional votes that cause v to become chosen at m, assuming that an acceptor a can cast new votes only in ballots numbered $\widehat{\beta}_a$ or higher.

Definition 3 A c-struct v is *choosable at* balnum m in ballot array β iff there exists an m-quorum Q such that $v \sqsubseteq \beta_a[m]$ for every acceptor a in Q with $\widehat{\beta}_a > m$.

It follows immediately from the definition that if v is chosen at m in β, then it is choosable at m in β.

We define v to be *safe at m* iff it is an extension of any value choosable at a balnum less than m, and we define β to be *safe* iff every $\beta_a[m]$ that is a c-struct is safe at m.

Definition 4 A c-struct v is *safe at m* in β iff $w \sqsubseteq v$ for every balnum $k < m$ and every c-struct w that is choosable at k. A ballot array β is *safe* iff for every acceptor a and balnum k, if $\beta_a[k]$ is a c-struct then it is safe at k in β.

Observe that if v is safe at m in β, then every extension of v is also safe at m in β. Every c-struct is trivially safe at 0 in any ballot array.

The following result shows that a voting algorithm can satisfy the consistency requirement of consensus by ensuring that an acceptor votes in each ballot number m only for c-structs that are safe at m. Detailed proofs of all the propositions in this section appear in Section B of the Appendix.

Proposition 1 If a ballot array β is safe, then the set of values that are chosen in β is compatible.

Proposition 1 shows that an algorithm can satisfy the consistency requirement of consensus by allowing acceptors to vote only for safe values. Satisfying liveness requires that the algorithm be able to get values chosen, which requires that acceptors must be able to vote in some ballot with a larger balnum than any in which they have already voted. Hence, the algorithm must be able to find values that are safe at some sufficiently large balnum. Moreover, since liveness should be satisfied if only a quorum of acceptors are non-faulty, the algorithm should be able to find safe values knowing only the states of acceptors in some quorum. Hence, we need to be able to compute safe values knowing only $\widehat{\beta}_a$ and the subarray β_a for the acceptors a in some quorum. We now show how this is done.

We define a set $\mathit{ProvedSafe}(Q, m, \beta)$ of c-structs that depends only on the values of $\beta_a[j]$ for $a \in Q$ and $j < m$. We then show that, if $\widehat{\beta}_a \geq m$ for every a in Q, then every c-struct in $\mathit{ProvedSafe}(Q, m, \beta)$ is safe at m in β. In the following definition, $\mathit{Max}\ B$ is the largest element in the finite set B of balnums.

Definition 5 For any balnum m, m-quorum Q, and ballot array β, let:

- $k \triangleq \mathit{Max}\ \{i \in \mathit{BalNum} \mid (i < m) \land (\exists a \in Q : \beta_a[i] \neq \mathit{none})\}$.
  [This set is non-empty if $m > 0$ because β is a ballot array.]
- $\mathcal{R} \triangleq \{R \in \mathit{Quorum}(k) \mid \forall a \in Q \cap R : \beta_a[k] \neq \mathit{none}\}$.
- $\gamma(R) \triangleq \sqcap \{\beta_a[k] \mid a \in Q \cap R\}$, for any R in $\mathcal{R}$.
  [$\gamma(R)$ exists by CS3 because Q and R are quorums, so $Q \cap R$ is non-empty.]
- $\Gamma \triangleq \{\gamma(R) \mid R \in \mathcal{R}\}$

Then $\mathit{ProvedSafe}(Q, m, \beta)$ is defined to equal

$$\begin{array}{l} \textsc{if}\ \mathcal{R} = \{\,\}\ \textsc{then}\ \{\beta_a[k] \mid (a \in Q) \land (\beta_a[k] \neq \mathit{none})\} \\ \qquad\qquad\quad \textsc{else}\ \ \textsc{if}\ \Gamma \text{ is compatible } \textsc{then}\ \{\sqcup \Gamma\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \textsc{else}\ \ \{\,\} \end{array}$$

The set Γ is finite because the set of acceptors is finite. Hence, if Γ is compatible, then $\sqcup \Gamma$ exists. If $\mathcal{R}$ is non-empty, then $\mathit{ProvedSafe}(Q, m, \beta)$ contains at most one element.

Proposition 2 For any balnum $m > 0$, m-quorum Q, and ballot array β, if β is safe and $\widehat{\beta}_a \geq m$ for all $a \in Q$, then every element of $ProvedSafe(Q, m, \beta)$ is safe at m in β.

This proposition implies that an algorithm can find a c-struct safe at a ballot number m by computing the set $ProvedSafe(Q, m, \beta)$ for an m-quorum Q, if that set is non-empty. Let k, $\mathcal{R}$, and Γ be as in the definition. The definition implies that $ProvedSafe(Q, m, \beta)$ is non-empty if $\mathcal{R}$ is empty. To find safe values, we must ensure that $ProvedSafe(Q, m, \beta)$ is also non-empty if $\mathcal{R}$ is non-empty. By the definition, this means showing that Γ is compatible if $\mathcal{R}$ is non-empty. The proof of Proposition 3 in the appendix shows that this is the case if k is a fast balnum. However, it need not be true for a classic balnum k. To ensure that $ProvedSafe(Q, m, \beta)$ is non-empty, we need β to satisfy another condition that we now define.

Definition 6 A ballot array β is called *conservative* iff for every classic balnum m and all acceptors a and b, if $\beta_a[m]$ and $\beta_b[m]$ are both different from *none*, then they are compatible.

Proposition 3 For any balnum $m > 0$, m-quorum Q, and ballot array β, if β is conservative then $ProvedSafe(Q, m, \beta)$ is non-empty.

5.3 The Abstract Algorithm

Paxos assumes some method of selecting a single leader. However, a unique leader is required only to ensure liveness. The safety requirements for consensus are satisfied even if there is no leader, or if multiple leaders are selected. We assume a set of possible leaders among which the leader is to be chosen. Each balnum is assigned to a possible leader, each possible leader being assigned unbounded sets of both classic and fast balnums.

Since we are considering only safety properties here, we don't care how many of the possible leaders actually are leaders. So, we drop the "possible" and simply call them leaders.

Intuitively, the abstract algorithm works as follows. Each acceptor participates in a sequence of ballots. It participates in only one ballot at a time, ending its participation in a ballot by joining a higher-numbered ballot. Voting in that ballot begins when the leader of a ballot (the one assigned its balnum) suggests c-structs for acceptors to vote for in that ballot. In a fast ballot, an acceptor first votes for a c-struct suggested by the leader and then decides by itself what additional c-structs to vote for. In a classic ballot, an acceptor votes only for c-structs suggested by the leader.

Acceptors, leaders, and learners perform the following actions. An acceptor a can at any time stop participating in its current ballot and join a new one with number m by performing action $JoinBallot(a, m)$. When enough acceptors have joined ballot m, its leader can perform a $StartBallot(m, Q)$ action to suggest a c-struct for that ballot. If this is a classic ballot, the leader then suggests additional c-structs by performing action $Suggest(m, C)$ for proposed commands C. This action suggests a new c-struct $v \bullet C$, where v is a c-struct the leader had previously suggested. An acceptor a can perform action $ClassicVote(a, v)$ to vote in its current ballot for a c-struct v suggested by the leader. If it is participating in a fast ballot and has already voted for a c-struct suggested by the leader, acceptor a can then perform action $FastVote(a, C)$ to vote for a new c-struct containing the proposed command C. It can keep performing $FastVote(a, C)$ actions for different commands C, until it joins a higher-numbered ballot. A learner l can at any time perform an $AbstractLearn(l, v)$ action that sets $learned[l]$ to v, if v is an extension of $learned[l]$ that is chosen.

We describe the algorithm precisely in terms of the following variables.

learned An array of c-structs, where $learned[l]$ is the c-struct currently learned by learner l. Initially, $learned[l] = \perp$ 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, $\widehat{bA}_a = 0$, $bA_a[0] = \perp$ and $bA_a[m] = none$ for all $m > 0$. (Every acceptor casts a default vote for $\perp$ in ballot 0, so the algorithm begins with $\perp$ chosen.)

minTried, *maxTried* Arrays, where $minTried[m]$ and $maxTried[m]$ are either both c-structs or both equal to *none*, for every balnum m. All the c-structs suggested thus far by the leader in ballot m are extensions of $minTried[m]$ and prefixes of $maxTried[m]$. Initially, $minTried[0] = maxTried[0] = \perp$ and $minTried[m] = maxTried[m] = none$ for all $m > 0$.

The algorithm maintains the following three invariants, where a c-struct v is said to be *proposed* iff it is an element of $Str(propCmd)$.

Tried **Invariant** For all balnums m,

1. $minTried[m] \sqsubseteq maxTried[m]$
2. If $minTried[m] \neq none$, then $minTried[m]$ is safe at m in bA and $maxTried[m]$ is proposed.

[This implies that $minTried[m]$ and $maxTried[m]$ are both proposed and are both safe at m in bA, if either is not $none$.]

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

1. $minTried[m] \sqsubseteq bA_a[m]$.
2. If m is a classic balnum, then $bA_a[m] \sqsubseteq maxTried[m]$.
   [This and part 2 of the *Tried* invariant imply that $bA_a[m]$ is proposed.]
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.

It is easy to check that the invariants are satisfied by the initial values of the variables. Observe that because the extension of a safe c-struct is safe, part 2 of the *Tried* invariant and part 1 of the bA invariant imply that bA is safe. The bA invariant implies that bA is also conservative.

We now show that these invariants imply that the algorithm satisfies the nontriviality and consistency requirements.

Nontriviality This is asserted by part 1 of the *learned* invariant.

Consistency By the bA invariant, $bA_a[m] \neq none$ implies $minTried[m] \sqsubseteq bA_a[m]$, which implies $minTried[m] \neq none$. Part 2 of the *Tried* invariant then implies that $bA_a[m]$ is safe at m in bA. This shows that bA is safe, so Proposition 1 implies that the set of values chosen in bA is compatible. Consistency then follows from part 2 of the *learned* invariant, since the $\sqcup$-completion of a compatible set is compatible.

To complete the description of the abstract algorithm, we now specify each of its atomic actions.

$Propose(C)$ 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.

$StartBallot(m, Q)$ for balnum m and m-quorum Q. It is enabled iff

- $maxTried[m] = none$ and
- $\forall a \in Q : \widehat{bA}_a \geq m$.

It sets $minTried[m]$ and $maxTried[m]$ to $w \bullet \sigma$ for an arbitrary element w in $ProvedSafe(Q, m, bA)$ and sequence σ in $Seq(propCmd)$.

$Suggest(m, C)$ for balnum m and command C. It is enabled iff

- $C \in propCmd$ and
- $maxTried[m] \neq none$.

It sets $maxTried[m]$ to $maxTried[m] \bullet C$.

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

- $maxTried[\widehat{bA}_a] \neq none$,
- $minTried[\widehat{bA}_a] \sqsubseteq v \sqsubseteq maxTried[\widehat{bA}_a]$, and
- $bA_a[\widehat{bA}_a] = none$ or $bA_a[\widehat{bA}_a] \sqsubset v$

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

$FastVote(a, C)$ 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 $learned[l] \sqcup v$.

Note that the *Join* and *StartBallot* actions are never enabled for balnum 0. (The initial state is one in which those ballot 0 actions have already been performed.)

We first need to show that these actions are type correct, meaning that they set the variables to values of the right type. The only non-trivial condition to check is that bA is always set to a ballot array. Since $bA_a[m]$ is changed only by setting it to a c-struct for $m = \widehat{bA}_a$, and $\widehat{bA}_a$ is only increased, this follows from the definition of *ballot array*.

We now show that each of these actions maintains the invariance of the three invariants. In the proofs, we let an ordinary expression *exp* be the value of that expression before executing the action, and *exp*$'$ be its value after the execution.

$Propose(C)$ This action only increases the set *propCmd*, and this is easily seen to preserve the invariants.

$JoinBallot(a, m)$ This action changes only $\widehat{bA}_a$, so it does not affect the bA or *learned* invariant. It could violate the *Tried* invariant only if $minTried[m]$ is safe at m in bA but not in bA', for some m. But the definition of *choosable at* implies that if w is choosable at k in bA', then it is choosable at k in bA. Hence, the definition of *safe at* implies that any c-struct v safe at m in bA is also safe at m in bA'.

$StartBallot(m, Q)$ This action changes only $minTried[m]$ and $maxTried[m]$, setting them from *none* to a c-struct v that is safe at m in bA by Proposition 2 and the observation that any extension of a safe c-struct is safe. The bA invariant, assumption CS3, and the definition of *ProvedSafe* imply that v is proposed. Hence, the action preserves the *Tried* invariant. It preserves the bA invariant because part 1 of that invariant implies $bA_a[m] = none$ for all acceptors a. It preserves the *learned* invariant because it does not change *learned*, *propCmd*, or what values are chosen in bA.

$Suggest(m, C)$ This action changes only $maxTried[m]$, setting it to an extension of its previous value. Part 2 of the *Tried* invariant implies that $maxTried[m]$ is safe and proposed. Since the extension of a safe value is safe, $maxTried[m]'$ is safe. Since the action is enabled only if C is proposed, $maxTried[m]'$ is also proposed. It is then easy to check that the invariants are maintained.

$ClassicVote(a, v)$ This action changes only $bA_a[\widehat{bA}_a]$, setting it to v. Since $minTried[\widehat{bA}_a] \sqsubseteq v \sqsubseteq maxTried[\widehat{bA}_a]$, the action clearly maintains the bA and *learned* invariants. It can violate the *Tried* invariant only by making $minTried[m]$ unsafe at m in bA' for some balnum m. But it follows from the definition of *choosable at* that any c-struct choosable at balnum k in bA' is choosable at k in bA, which implies that the action preserves safety at any balnum m.

$FastVote(a, C)$ This action changes only $bA_a[\widehat{bA}_a]$, setting it to an extension of its previous value. Since it is performed only if $\widehat{bA}_a$ is a fast balnum, it obviously preserves the bA invariant. It preserves the *Tried* invariant for the same reason that the *ClassicVote* action does, and it is easily seen to preserve the *learned* invariant.

$AbstractLearn(l, v)$ This action trivially maintains the *Tried* and bA invariants. The enabling condition implies that it maintains part 2 of the *learned* invariant. Part 1 of that invariant follows from parts 2 and 3 of the bA invariant and assumption CS3, which imply that any chosen c-struct is proposed.

Since the invariants hold in the initial state, this proves that they hold throughout any execution of the algorithm. As observed above, this proves that the algorithm satisfies the non-triviality and consistency requirements. Since $learned[l]$ is changed only by action $AbstractLearn(l, v)$, which sets it to $learned[l] \sqcup v$, the stability requirement is obviously satisfied.

5.4 A Distributed Abstract Algorithm

The abstract algorithm is non-distributed because an action performed by one process depends on the values of variables set by another process. For example, the action $ClassicVote(a, v)$ performed by acceptor a depends upon the values of $minTried$ and $maxTried$, variables set by a leader. To implement this non-distributed algorithm with a distributed one, we must have each process send information to other processes when it changes the values of its variables. For example, the leader of ballot m sends messages to acceptors when it changes $minTried[m]$ or $maxTried[m]$. The problem is that the values of those variables can change between when the message is sent and when it is received. This problem is solvable because the values of variables change monotonically. The values of $maxTried[m]$ and $bA_a[m]$ change only from $none$ to a sequence of c-structs, each a prefix of the next. The value of $minTried[m]$ changes only from $none$ to a c-struct, and then remains unchanged. Each $\widehat{bA}_a$ can only increase.

We now describe a distributed abstract algorithm. Its variables consist of the variables of the abstract algorithm plus additional variables that we don't specify that represent what messages are in transit and what messages have been received. The variables taken from the abstract algorithm are initialized as in that algorithm, and initially no messages have been sent. We now describe each of the distributed algorithm's actions, and we explain what action of the abstract algorithm it implements.

Following previous descriptions of classic Paxos [5], we divide acceptor and leader actions into phase 1 and phase 2 actions. Phase 1 actions implement the *Join* and *StartBallot* actions of the abstract algorithm; phase 2 actions implement *Suggest*, *ClassicVote*, and *FastVote*. (The *FastVote* action does not appear in classic Paxos.) Phase 1 actions are never enabled for ballot 0. (They are unnecessary.)

$SendProposal(C)$ executed by the proposer of command C. The action is always enabled. It sets $propCmd$ to $propCmd \cup \{C\}$ and sends a

$\langle$ "propose", $C\rangle$ message. This message may be sent to one or more possible leaders and/or to the acceptors; we discuss later where they are sent.

The action implements the abstract algorithm's $Propose(C)$ action if $C \notin propCmd$; otherwise it leaves that algorithm's variables unchanged.

$Phase1a(m)$ executed by the leader of ballot numbered m. The action is enabled iff $maxTried[m] = none$. It sends the message $\langle$ "1a", $m\rangle$ to acceptors.

The action leaves the abstract algorithm's variables unchanged.

$Phase1b(a, m)$ executed by acceptor a, for balnum m. The action is enabled iff a has received a $\langle$ "1a", $m\rangle$ message (from the leader) and $\widehat{bA}_a \leq m$. It sends the message $\langle$ "1b", m, a, $bA_a\rangle$ to the leader and sets $\widehat{bA}_a$ to m.

This action implements the abstract algorithm's $JoinBallot(a, m)$ action.

$Phase2Start(m, v)$ executed by the leader of ballot m, for c-struct v. The action is enabled when:

- $maxTried[m] = none$,
- the leader has received a "1b" message for balnum m from every acceptor in an m-quorum Q, and
- $v = w \bullet \sigma$, 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, $\widehat{\beta}_a = k$ and the leader has received a message $\langle$ "1b", m, a, $\rho\rangle$ with $\beta_a = \rho$.

The action sets $minTried[m]$ and $maxTried[m]$ to $v \bullet \sigma$ and sends the message $\langle$ "2a", m, $v \bullet \sigma\rangle$ to acceptors, where σ is some sequence of commands, each element of which the leader has received in "propose" messages.

We now show that this action implements the abstract algorithm's $StartBallot(m, Q)$ action. To show that the enabling condition implies that $StartBallot(m, Q)$ is enabled, we must show that $\widehat{bA}_a \geq m$ for all $a \in Q$. This is true because the $Phase1b$ message that sent the "1b" message from acceptor $a \in Q$ set $\widehat{bA}_a$ to m, and the value of $\widehat{bA}_a$ never decreases. Since a "propose" message is send only for commands in *propCmd*, it is clear that the action implements $StartBallot(m, Q)$ if $ProvedSafe(Q, m, \beta)$ equals $ProvedSafe(Q, m, bA)$. These *ProvedSafe* sets are equal for the following reason. The set $ProvedSafe(Q, m, \beta)$ depends only on the values $\beta_a[k]$ for $a \in Q$ and $k < m$. But $\beta_a[k]$ equals $\rho[k]$ for some $\langle \text{"1b"}, m, a, \rho \rangle$ message sent by a. When that message was sent, $\rho[k]$ equaled $bA_a[k]$. Moreover, the $Phase1b$ action that sent the message also set $\widehat{bA}_a$ to m, which prevents any further change to $bA_a[k]$ for $k < m$. Hence, $\beta_a[k]$ equals the current value of $bA_a[k]$ for all $a \in Q$ and $k < m$, so $ProvedSafe(Q, m, \beta) = ProvedSafe(Q, m, bA)$. This completes the proof that the action implements $StartBallot(m, Q)$.

$Phase2aClassic(m, C)$ executed by the leader of ballot m, for command C. The action is enabled iff $maxTried[m] \neq none$ and the leader has received a $\langle \text{"propose"}, C \rangle$ message. It sends the message

$$\langle \text{"2a"}, m, maxTried[m] \bullet C \rangle$$

to the acceptors and sets $maxTried[m]$ to $maxTried[m] \bullet C$.

It is easy to see that this implements action $Suggest(m, C)$ of the abstract algorithm.

$Phase2bClassic(a, m, v)$ executed by acceptor a for balnum m and c-struct v. The action is enabled iff $\widehat{bA}_a = m$, acceptor a has received the message $\langle \text{"2a"}, m, v \rangle$, and $bA_a[m]$ equals *none* or $bA_a[m] \sqsubset v$. It sets $bA_a[m]$ to v and sends a $\langle \text{"2b"}, m, a, v \rangle$ message to every learner.

This action clearly implements action $ClassicVote(a, v)$ of the abstract algorithm if $minTried[m] \sqsubseteq v \sqsubseteq maxTried[m]$. But $minTried[m] \sqsubseteq v = maxTried[m]$ held when the "2a" message was sent, $minTried[m]$ never changes once it is different from *none*, and $maxTried[m]$ can only increase. Hence, $minTried[m] \sqsubseteq v \sqsubseteq maxTried[m]$ must hold.

$Phase2bFast(a, m, C)$ executed by acceptor a for balnum m and command C. The action is enabled iff m is a fast balnum, $\widehat{bA}_a = m$, $bA_a[m] \neq none$, and a has received a $\langle \text{"propose"}, C \rangle$ message. It sets $bA_a[m]$ to $bA_a[m] \bullet C$ and sends the message $\langle \text{"2b"}, m, a, bA_a[m] \bullet C \rangle$ to every learner.

It is easy to see that this implements action $FastVote(a, C)$ of the abstract algorithm.

Learn(l, v) performed by learner l for c-struct v. The action is enabled iff, for some balnum m and some m-quorum Q, learner l has received a message $\langle$ "2b", m, a, $w\rangle$ with $v \sqsubseteq w$ from every acceptor a in Q. It sets $learned[l]$ to $learned[l] \sqcup v$.

The existence of the message $\langle$ "2b", m, a, $w\rangle$ implies $w \sqsubseteq bA_a[m]$, since the value of $bA_a[m]$ equaled w when the message was sent and can only increase. Hence, the enabling condition implies that v is chosen in bA, so this action implements action $AbstractLearn(l, v)$ of the abstract algorithm.

Since the distributed abstract algorithm implements the abstract algorithm, it satisfies the nontriviality, consistency, and stability requirements.

5.5 The Generalized Paxos Consensus Algorithm

In our distributed abstract algorithm, processes maintain a lot of information. A leader maintains the values of $minTried[m]$ and $maxTried[m]$ for each of its balnums m; an acceptor a maintains the array bA_a. Moreover, the complete array bA_a is sent in phase 1b messages. We obtain the generalized Paxos consensus algorithm by eliminating most of this information, so a process maintains only the data it needs.

The variable *minTried* is not used at all by the algorithm, so it can be eliminated. (It appears only in the proof that the distributed algorithm's actions implement the non-distributed abstract algorithm's actions.) Moreover, when the leader has begun the execution of ballot m with a $Phase1a(m)$ action, it can forget about lower-numbered ballots. (It can ignore phase 1b messages for lower-numbered ballots.) Therefore, a leader r need only maintain a variable $maxStarted[r]$ whose value is the largest balnum m assigned to it such that $maxTried[m] \neq none$, and a variable $maxVal[r]$ whose value is $maxTried[maxStarted[r]]$.

To compute $ProvedSafe(Q, m, bA)$, a leader does not need the complete arrays bA_a; it needs to know only the largest balnum $k < m$ such that $bA_a[k] \neq none$ and the value of $bA_a[k]$. It is therefore not hard to see that an acceptor a need keep only the following information:

$$\begin{aligned} mbal[a] &\triangleq \widehat{bA}_a \\ bal[a] &\triangleq Max\{k \in balnum \mid bA_a[k] \neq none\} \\ val[a] &\triangleq bA_a[bal[a]] \end{aligned}$$

The $Phase1b(a, m)$ action is then enabled iff a has received a $\langle$ "1a", $m\rangle$ message with $m < mbal[a]$; the action sets $mbal[a]$ to m and sends the message $\langle$ "1b", m, a, $bal[a]$, $val[a]\rangle$. The modifications to the other actions are straightforward. Since this is just an optimization of the distributed abstract algorithm, it satisfies the safety requirements for consensus.

In the generalized Paxos consensus algorithm, the state of a leader, acceptor, or learner consists of a c-struct and at most two balnums. The largest message sent (a phase 1b message) contains a c-struct, a pair of balnums, and the name of an acceptor. This is very little data, if c-structs are small. However, a c-struct might be a history containing all the commands ever executed by the system. It would be impractical to send a lot of messages containing such c-structs. It might even be difficult for a process to store a c-struct. The problem of handling large c-structs is discussed below in Section 6.3.

6 Implementation Considerations

6.1 Normal Operation

In the usual implementations, one builds a system with some number N of processors acting as acceptors. One can then let a quorum for both classic and fast balnums consist of any set with at least $\lfloor 2N/3 \rfloor + 1$ acceptors. One can also let a quorum for classic balnums consist of any set of at least $\lfloor N/2 \rfloor + 1$ acceptors, and let a quorum for fast balnums consist of any set of at least $\lceil 3N/4 \rceil$ acceptors. A little simple set theory shows that the Quorum Assumption is satisfied by these choices of quorums.

We consider how the system behaves in normal operation, starting with the failure of the current leader and the selection of a new one. As part of the leader-selection process, the new leader tries to learn what acceptors are working. It also tries to learn the number of the largest ballot that was in progress and chooses a larger balnum m that is assigned to it. The leader chooses a fast balnum m if $\lfloor 2N/3 \rfloor + 1$ acceptors are working; otherwise, it must choose a classic one. It then begins ballot m by executing its $Phase1a(m)$ action, sending phase 1a messages to an m-quorum of acceptors. Upon receipt of those messages, the acceptors execute $Phase1b$ actions and send phase 1b messages to the leader. (Those $Phase1b$ actions are enabled unless an acceptor had already participated in a higher-numbered ballot, in which case the acceptor notifies the leader and the leader tries again with a larger balnum.)

When the leader has received phase 1b messages from an m-quorum, it begins the second phase of ballot m (the voting phase) by executing a $Phase2aStart$ action, sending phase 2a messages to acceptors in an m-

quorum. Those acceptors will then execute their *Phase*2*aClassic* action, sending phase 2b messages to the learners. The effect of this is to complete the choosing of all c-structs that failed to be chosen in an earlier ballot. (The failure of the previous leader may have resulted in a partially completed ballot, in which fewer than a quorum of acceptors voted for some c-struct.)

What happens next depends on whether the leader has chosen a classic or a fast balnum. If it has chosen a classic balnum, then it notifies proposers to send it their "propose" messages. Upon receipt of such a message, it executes the *Phase*2*aClassic* action, sending phase 2a messages to an m-quorum of acceptors. Upon receipt of a phase 2a message, acceptors execute *Phase*2*bClassic* actions, sending phase 2b messages to the learners. A learner learns a c-struct containing the proposed command when it has received the phase 2b messages from an m-quorum. Thus, three message delays elapse between the proposal of a command and the learning of a c-struct containing that command. Approximate Theorem 3 implies that this delay is optimal if fewer than $\lfloor 2N/3 \rfloor + 1$ acceptors are non-faulty.

If the leader has chosen a fast balnum, then it notifies proposers to send their proposals directly to an m-quorum of acceptors. Upon receipt of a proposal, an acceptor executes a *Phase*2*bFast* action, sending a phase 2b message to the learners. By assumption CS4, a learner learns a c-struct containing a command C if it receives phase 2b messages with compatible c-structs containing C from an m-quorum of acceptors. The command is therefore learned within two message delays of its proposal, if a quorum of acceptors all send compatible c-structs. When the c-structs are histories, this will be the case unless an interfering command D is proposed concurrently. In that case, some acceptors a may execute $Phase2bFast(a, m, C)$ before executing $Phase2bFast(a, m, D)$, and other acceptors may execute the actions in the opposite order. Thus, some acceptors may vote for a c-struct $w \bullet C \bullet D$ and others for the incompatible c-struct $w \bullet D \bullet C$, for some c-struct w. In that case, no c-struct containing either C or D is chosen in ballot m.

When such a collision occurs, the leader can intervene by executing a $Phase1a(n)$ action for a higher fast balnum n. Suppose as a result of the phase 1b messages it receives, the leader's $Phase2Start(n, w)$ action is enabled. It begins phase 2 of ballot n by executing that action, sending phase 2a messages for the c-struct $w \bullet C \bullet D$ or $w \bullet D \bullet C$. Upon receipt of this message, the acceptors perform the corresponding *Phase*2*bClassic* actions, sending phase 2b messages that cause a c-struct containing C and D to be learned. The acceptors then resume normal fast operation, receiving proposals directly from proposers and executing *Phase*2*bFast* actions.

The failure or repair of an acceptor can cause the leader to switch from fast to slow Paxos or vice-versa. It does this by executing $Phase1a(n)$ for a new balnum n.

6.2 Ensuring Liveness

We now consider the liveness condition LiveChoice(C, l). It holds under certain assumptions that can be stated in terms of the concept of a nonfaulty set of processes. Intuitively, a set of processes is nonfaulty iff there are upper bounds on the time taken by each process to perform an action and on the delivery time of messages sent from one of the processes to another. We do not attempt to define precisely what nonfaulty means.

When classic balnums are used, the liveness properties of generalized Paxos are essentially the same as for ordinary Paxos. It is not hard to see that progress can be guaranteed if eventually:

- A single unique, non-faulty leader r is chosen, and no other possible leader performs $Phase1a$ actions.
- Leader r executes a $Phase1a(m)$ action for a sufficiently large classic balnum m, and executes no $Phase1a(n)$ actions for $n > m$.
- All messages sent between r and an m-quorum Q of acceptors are eventually delivered.
- Leader r and all acceptors in Q eventually execute enabled Phase1 and Phase2 actions.

Under those assumptions, condition LiveChoice(C, l) will hold if the proposer of C eventually executes a $SendProposal(C)$ action, the message $\langle$ "propose", $C \rangle$ sent by that action is received by leader r, and learner l receives the phase 2b messages sent by acceptors in Q. The details are the same as for ordinary Paxos, and the reader is referred to the proof by de Prisco et al. [2].

To achieve liveness with fast Paxos, the leader must receive phase 2b messages and observe if a collision has occurred—a situation indicated by receipt of phase 2b messages for incompatible c-structs. It must then start a new, higher-numbered ballot and get the conflicting proposed commands chosen as described above. The basic idea is clear, though formalizing the details is tedious.

6.3 Large C-Structs

The generalized Paxos algorithm uses c-structs throughout, saving their values in variables and sending them in messages. This is obviously a problem if c-structs are large—for example, if they contain the entire execution history of the system. We now indicate how they are handled in practice.

A process constructs a c-struct v by appending a short command sequence σ to another c-struct w. When a process sends v in a message, it has already sent w to the same recipient. So it can send v by sending only σ and the identifier of the message containing w. Sending c-structs in messages therefore poses no problem.

If a c-struct contains the entire history of the system, even storing it in memory may be a problem. Moreover, a leader must compute glbs and lubs of sets of c-structs to execute a *Phase2Start* action.

Maintaining the entire execution history is a problem faced by the ordinary state-machine approach. Even if the entire command sequence can be kept in a process's memory, restarting a failed server could require sending an infeasibly large sequence of commands. This problem is solved by letting a process forget the initial prefix of the command sequence, remembering only the state after that prefix's execution. That state is the only information needed to execute later commands. A process thus remembers the state after executing some initial prefix of commands, the number of commands in that prefix, and the sequence of subsequent commands.

The same idea can be applied in general when c-structs are histories. If a server has learned that the "current" history of the system is $v \bullet \sigma$ for some c-seq σ, then it can execute the commands in σ knowing only the state after executing the history v. So in generalized Paxos, servers can also forget prefixes of the current history. However, we now explain why this is not as simple for arbitrary c-structs as it is for command sequences.

To execute the generalized Paxos algorithm, a process must be able to compute the glb and lub of two c-structs. For example, to execute action $Phase2bClassic(a, m, v)$ upon receipt of a phase 2a message containing c-struct v, acceptor a must check that $bA_a[m] \sqsubseteq v$, which is equivalent to checking that $v = bA_a[m] \sqcup v$. Suppose that a process a must compute the lub of a c-struct v_a in its memory and a c-struct v_b that it has received in a message from another process b. If prefixes of c-structs have been forgotten, then a will know only that $v_a = w_a \bullet \sigma_a$ and $v_b = w_b \bullet \sigma_b$ for known c-seqs σ_a and σ_b, but for prefixes w_a and w_b that it has forgotten. In general, it is not possible to compute $v_a \sqcup v_b$ knowing only σ_a, σ_b, and the states after executing the histories w_a and w_b.

To solve this problem, we introduce the concept of a *checkpoint*. A checkpoint is a command C satisfying the following property: for any c-seqs ρ, σ, and τ, if $\bot \bullet \rho \bullet C \bullet \sigma = \bot \bullet \tau$, then there is a c-seq η such that $\tau = \eta \circ \langle C \rangle \circ \sigma$ and $\bot \bullet \rho = \bot \bullet \eta$. For a monoidal c-struct set, this implies that any c-struct v can be written uniquely in the form $v_1 \bullet C \bullet \cdots \bullet C \bullet v_n$ where each v_i equals $\bot \bullet \tau_i$ and the c-seq τ_i does not contain the command C. For histories, a checkpoint is any command that interferes with every command. Any state machine can be augmented by a checkpoint that is defined to be a no-op (produces no output and leaves the state unchanged) that interferes with ever command.

Assume a special checkpoint command C. A leader can periodically propose command C. A prefix is forgotten only if it is of the form $v \bullet C$. In an ordinary state-machine implementation, a process might remember only the state after executing command number i, for some i, and the sequence of later commands. Similarly, in an implementation of generalized Paxos, a process might remember only the state after executing the prefix v_i ending with the i^{th} checkpoint and the c-struct w such that the history it currently knows is $v_i \bullet w$.

Just as in the ordinary state-machine approach, an implementation can use a sequence of separate instances of the generalized Paxos algorithm to choose successive parts of the command history. In the ordinary approach, the command sequence $C_1, C_2, \ldots$ is chosen by letting the i^{th} instance of the ordinary Paxos consensus algorithm choose the command C_i. In generalized Paxos, a command history $v_1 \bullet C \bullet v_2 \bullet C \bullet \cdots$ can be chosen by letting the i^{th} instance of the generalized Paxos algorithm choose the history $v_i \bullet C$, where C is a special checkpoint command. The procedures for forgetting history prefixes and updating restarted servers in generalized Paxos are then completely analogous to the ones for the ordinary state-machine method.

7 Summary

Classical Paxos uses a sequence of consensus algorithms to choose a sequence of commands. In normal operation, a client (proposer) sends its command to the leader, which forwards it in a phase 2a message to the acceptors, which then send phase 2b messages to the servers (learners), which execute the command upon receipt of enough phase 2b messages. Thus, it takes three message delays for a command to be executed. Messages can be saved at the cost of an extra message delay by having the phase 2b messages sent only to the leader.

Consistency of the system is maintained despite any number of non-Byzantine failures. To ensure progress despite the failure of F nodes requires more than $2F$ acceptors.

Fast Paxos saves one message delay by having the client send its command directly to the acceptors. Allowing fast progress despite the failure of E nodes requires more than $2E + F$ acceptors. However, if two clients concurrently send commands, then the normal procedure might fail to choose a command, incurring one or more extra message delays.

Instead of executing a sequence of ordinary consensus algorithms, each choosing a single command, we have restated the problem of implementing a state machine as that of agreeing on a growing command history. We generalized both ordinary consensus and consensus on command histories to the problem of learning a monotonic sequence of objects called command structures, and we generalized the Paxos consensus algorithm to solve this problem.

The purpose of this generalization is to obtain an algorithm with the same message delay as fast Paxos, but that remains fast despite concurrent issuing of client commands, if those commands are non-interfering. In many applications, concurrently issued commands are almost always non-interfering. The generalized Paxos algorithm provides a new method of implementing such systems that, in the normal case, is optimal in terms of the number of message delays required to execute a command.

In principle, a single instance of the generalized consensus algorithm can be used to implement a system. In practice, a sequence of separate instances will be used, each choosing the portion of the history between two successive checkpoints.

All the implementation details of the ordinary state-machine approach apply to the generalized algorithm. In particular, reconfiguration can be performed by state-machine commands. In the original state-machine approach, based on a sequence of instances of a consensus algorithm, the set of acceptors (and hence the set of quorums) used in instance i can be determined by the state after executing command $i - 1$.[1] The same applies to the generalized state-machine approach based on command histories. The instance of generalized Paxos used to choose the portion of the history between checkpoints i and $i+1$ can be determined by the state after executing checkpoint i

[1] In ordinary Paxos, one allows pipelining of $\alpha - 1$ instances by letting the set of acceptors in instance i depend on the state after command $i - \alpha$. There is no reason to do this in generalized Paxos, where each instance chooses a set of commands.

Acknowledgments

Lasaro Jonas Camargos and Rodrigo Schmidt found some minor errors in an earlier version.

References

[1] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[2] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243:35–91, 2000.

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

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

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

[6] Leslie Lamport. Lower bounds for asynchronous consensus. In André Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[7] Leslie Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-71, Microsoft Research, July 2004. Currently available from `http://research.microsoft.com/users/lamport/pubs/pubs.html`, or by searching the Web for the 23-letter string obtained by removing the - characters from all-lamports-pubsonthe-web.

[8] A. Mazurkiewicz. Semantics of concurrent systems: A modular fixed point trace approach. In G. Rozenberg, editor, *FAdvances in Petri Nets 1984*, volume 188 of *Lecture Notes in Computer Science*, pages 353–375. Springer-Verlag, 1984.

[9] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast. *Distributed Computing*, 15(2):97–107, 2002.

A Lower-Bound Proof Ideas

We describe the ideas behind the proofs of the approximate theorems asserted in Section 2.3. We make no attempt to convince the reader that the results are really valid. In fact, most of them are false as stated. Their precise statements and rigorous proofs appear in [7].

Approximate Theorem 1 *Any two quorums have non-empty intersection.*

Proof Idea: We assume that Q_1 and Q_2 are disjoint quorums and obtain a contradiction. Let p_1 and p_2 be two proposers that propose different values C_1 and C_2, and let l_1 and l_2 be different learners. Suppose both of the sets $S_i \triangleq Q_i \cup \{p_i, l_i\}$ are nonfaulty, but all messages sent between S_1 and S_2 are lost. The requirement Liveness(C_i, l_i) implies that both learners l_i must learn a value. Nontriviality implies that l_i must learn v_i. Thus, l_1 and l_2 must learn different values, violating the consistency requirement. □

Approximate Theorem 2 *Learning is impossible in fewer than 2 message delays.*

Proof Idea: Suppose l learns a value proposed by p in one message delay. Then it is possible that every message sent by p was lost except for ones received by l. If all messages from p and l to other processes are lost, then there is nothing to prevent another learner from learning a different value proposed by another proposer, violating consistency. □

Approximate Theorem 3 *If Q_1 and Q_2 are fast quorums and Q is a quorum, then $Q_1 \cap Q_2 \cap Q$ is non-empty.*

Proof Idea: Let A_1 and A_2 be sets of acceptors such that A_1, A_2, and $Q_1 \cap Q_2$ are pairwise disjoint and $Q_i = A_i \cup (Q_1 \cap Q_2)$, for each i. Let p_1 and p_2 be proposers and let l_a be a learner. Let $\mathcal{F}_i$ be an execution in which p_i proposes a value v_i and sends messages to the acceptors in Q_i and to l_a, then the acceptors in Q_i then send messages to l_a, and l_a learns v_i. (Any messages sent by or to processes not in $Q_i \cup \{p_i, l_a\}$ are lost.) Define $\hat{i}$ so $\hat{1} = 2$ and $\hat{2} = 1$. (Thus, Q_i and $A_{\hat{i}}$ are disjoint, for each i.)

We now define two executions, $\mathcal{E}_1$ and $\mathcal{E}_2$ as follows. In both executions, p_1 and p_2 propose two different values v_1 and v_2. In $\mathcal{E}_i$, messages sent between processes in the set $Q_i \cup \{p_i, l_a\}$ are delivered very quickly, as are messages sent between processes in the set $A_{\hat{i}} \cup \{p_{\hat{i}}\}$; however messages sent between those two sets travel very slowly. Moreover, messages sent between processes in $Q_i \cup \{p_i, l_a\}$ are delivered so that the beginning of that execution looks to those processes exactly like the execution $\mathcal{F}_i$. Hence, l_a learns v_i in execution $\mathcal{E}_i$.

Suppose that l_a and all the acceptors in $Q_1 \cap Q_2$ lose communication with the rest of the processes. Executions $\mathcal{E}_1$ and $\mathcal{E}_2$ appear the same to those other processes—that is, to processes p_1 and p_2 and the acceptors in A_1 and A_2. Hence, those processes have no way of knowing if l_a learned v_1 or v_2. If there were a quorum Q disjoint from $Q_1 \cap Q_2$, then liveness requires that a different learner l_b eventually learn a value. This is impossible, since consistency cannot be ensured without knowing which value l_a learned. $\Box$

Approximate Theorem 4 *If, for every acceptor a, there is a quorum not containing a, then a consensus algorithm cannot ensure that, in the absence of failures, every learner learns a value in two message delays.*

Proof Idea: Let p_1 and p_2 be proposers and l_a a learner. Let $\mathcal{F}_i$ be a scenario in which p_i proposes value v_i, it sends messages to all learners and acceptors, all acceptors send messages to all learners, and l_a then learns v_i. Define $\hat{\imath}$ so $\hat{1} = 2$ and $\hat{2} = 1$. Let $\mathcal{E}_i$ be the scenario obtained from $\mathcal{F}_i$ by having all messages lost except those needed for $\mathcal{E}_i$ to look the same to l_a as $\mathcal{F}_i$, and having proposer $p_{\hat{\imath}}$ also proposes value $v_{\hat{\imath}}$, but letting all its messages arrive after all the messages that were in $\mathcal{F}_i$.

Let $a_1, \ldots, a_n$ be the acceptors, and define a sequence of executions $\mathcal{G}_0, \ldots, \mathcal{G}_n$ as follows. Let $\mathcal{G}_0$ equal $\mathcal{E}_1$ and $\mathcal{G}_n$ equal $\mathcal{E}_2$. In $\mathcal{G}_0$, all the messages sent by p_1 arrive before all the messages sent by p_2, while in $\mathcal{G}_n$, those messages all arrive in the opposite order. For $0 < i < n$, let $\mathcal{G}_i$ be the same as $\mathcal{G}_{i-1}$, except that the message from p_2 arrives at acceptor a_i before the message from p_1. Learner l_a learns value v_1 in $\mathcal{G}_0$ and v_2 in $\mathcal{G}_n$. So there is some $i > 0$ so that l_a learns v_1 in $\mathcal{G}_{i-1}$ and v_2 in $\mathcal{G}_i$.

Now consider an execution that begins like either $\mathcal{G}_{i-1}$ or $\mathcal{G}_i$, and then a_i and l_a both lose contact with the remaining processes. Both executions look exactly the same to those remaining processes, which therefore cannot tell which value l_a learned. There is a quorum Q not containing a_i all of whose processes have not failed, so another learner l_b must be able to learn a value. But there is no way to discover what value l_a has learned, so there is no way to ensure consistency while allowing l_b to learn a value. $\Box$.

B Proofs of Propositions

Proposition 1 If a ballot array β is safe, then the set of values that are chosen in β is compatible.

PROOF: By the definition of *chosen in* and *compatible*, it suffices to assume

1. β is safe
2. c-struct v is chosen at balnum m in β
3. c-struct w is chosen at balnum n in β

and to prove v and w are compatible.

1. Choose an m-quorum Q_v and an n-quorum Q_w such that $v \sqsubseteq \beta_a[m]$ for all $a \in Q_v$ and $w \sqsubseteq \beta_a[n]$ for all $a \in Q_w$.
   PROOF: Q_v and Q_w exist by assumptions 2 and 3 and the definition of *chosen at*.
2. CASE: $m = n$
   2.1. Choose an acceptor a in $Q_v \cap Q_w$.
   PROOF: a exists by the case assumption, step 1 (which implies Q_v and Q_w are m-quorums), and the Quorum Assumption.
   2.2. Q.E.D.
   PROOF: Steps 1 and 2.1 imply $v \sqsubseteq \beta_a[m]$ and $w \sqsubseteq \beta_a[m]$, so v and w are compatible.
3. CASE: $m < n$
   3.1. v is choosable at m in β.
   PROOF: By assumption 2, since *choosable at* implies *chosen at*.
   3.2. Choose an acceptor a in Q_w.
   PROOF: a exists by choice of Q_w (step 1), since the Quorum Assumption implies that any n-quorum is non-empty.
   3.3. Q.E.D.
   PROOF: Steps 3.2 and 1 imply $w \sqsubseteq \beta_a[n]$. Assumption 1 implies $\beta_a[n]$ is safe at n, so 3.1, the case assumption $m < n$, and the definition of *safe at* imply $v \sqsubseteq \beta_a[n]$. Hence, w and v are compatible.
4. Q.E.D.
   PROOF: By steps 2 and 3, the case $n < m$ following from 3 by symmetry.

Proposition 2 For any balnum $m > 0$, m-quorum Q, and ballot array β, if β is safe and $\widehat{\beta}_a \geq m$ for all $a \in Q$, then every element of $\mathit{ProvedSafe}(Q, m, \beta)$ is safe at m in β.

PROOF: Assume β is safe, $\forall a \in Q : \widehat{\beta}_a \geq m$, and $v \in \mathit{ProvedSafe}(Q, m, \beta)$. Let w be a c-struct choosable at some balnum $j < m$. By definition of *safe at*, it suffices to prove $w \sqsubseteq v$. Let k, $\mathcal{R}$, and $\gamma(R)$ be defined as in Definition 5.

1. Choose a j-quorum Q_w such that $\beta_a[j] \neq none$ and $w \sqsubseteq \beta_a[j]$ for all a in Q_w such that $\widehat{\beta}_a > j$.
   PROOF: Q_w exists by the assumption that w is choosable at j and the definition of *choosable at*.
2. $j \leq k < m$
   2.1. $k < m$ and $\beta_a[i] = none$ for all $a \in Q$ and all i with $k < i < m$.
   PROOF: By definition of k.
   2.2. Choose a in $Q \cap Q_w$.
   PROOF: a exists by step 1 (Q_w a j-quorum), the assumption that Q is an m-quorum, and the Quorum Assumption.
   2.3. $\widehat{\beta}_a \geq m$ and $\beta_a[i] = none$ for all i with $k < i < m$.
   PROOF: The hypothesis $\forall a \in Q : \widehat{\beta}_a \geq m$ and the choice of a (step 2.2) imply $\widehat{\beta}_a \geq m$. Step 2.1 implies $\beta_a[i] = none$ if $k < i < m$.
   2.4. $\beta_a[j] \neq none$
   PROOF: Step 2.3 and the hypothesis $j < m$ imply $\widehat{\beta}_a > j$. Step 2.2 (which implies $a \in Q_w$) and step 1 then imply $\beta_a[j] \neq none$.
   2.5. Q.E.D.
   PROOF: Steps 2.3 and 2.4 and the hypothesis $j < m$ imply $j \leq k$; step 2.1 asserts $k < m$.
3. CASE: $j = k$
   3.1. $Q_w \in \mathcal{R}$
   PROOF: The hypothesis $\forall a \in Q : \widehat{\beta}_a \geq m$ and step 2 ($j < m$) imply $\widehat{\beta}_a > j$ for all $a \in Q$. By step 1 and the case assumption $j = k$, this implies $\beta_a[k] \neq none$ for all $a \in Q \cap Q_w$.
   3.2. $\gamma(Q_w) \sqsubseteq v$
   PROOF: By step 3.1, the hypothesis $v \in ProvedSafe(Q, m, \beta)$, and the definition of *ProvedSafe*.
   3.3. $w \sqsubseteq \gamma(Q_w)$
   PROOF: Steps 1 and 3.1, the case assumption $j = k$, and the definition of $\mathcal{R}$ imply $w \sqsubseteq \beta_a[j]$ for all $a \in Q \cap Q_w$. The definitions of γ and of the glb imply $w \sqsubseteq \gamma(Q_w)$
   3.4. Q.E.D.
   PROOF: Assumption CS2 ($\sqsubseteq$ a partial order) and steps 3.2 and 3.3 imply $w \sqsubseteq v$.
4. CASE: $j < k$
   4.1. CASE: $\mathcal{R}$ is empty.
   4.1.1. Choose a in Q such that $\beta_a[k] \neq none$ and $v = \beta_a[k]$.

PROOF: a exists by case assumption 4.1 ($\mathcal{R}$ empty) and the hypothesis $v \in \mathit{ProvedSafe}(Q, m, \beta)$.

4.1.2. $w \sqsubseteq \beta_a[k]$

PROOF: Step 4.1.1 ($\beta_a[k] \neq \mathit{none}$), the hypotheses that w is choosable at j, case assumption 4 ($j < k$), and the assumption that β is safe.

4.1.3. Q.E.D.

PROOF: Steps 4.1.1 and 4.1.2 imply $w \sqsubseteq v$.

4.2. CASE: $\mathcal{R}$ is non$-$empty.

4.2.1. Choose a k-quorum R in $\mathcal{R}$.

PROOF: R exists by case assumption 4.2.

4.2.2. $w \sqsubseteq \beta_k[a]$ for all $a \in Q \cap R$

PROOF: For all $a \in Q \cap R$, step 4.2.1 and the definition of $\mathcal{R}$ imply $\beta_k[a] \neq \mathit{none}$. The hypothesis that w is choosable at j, case assumption 4 ($j < k$), and the hypothesis that β is safe then imply $w \sqsubseteq \beta_k[a]$.

4.2.3. $w \sqsubseteq \gamma(R)$

PROOF: Step 4.2.2, the definition of γ, and the definition of the glb.

4.2.4. $\gamma(R) \sqsubseteq v$

PROOF: Step 4.2.1, the hypothesis $v \in \mathit{ProvedSafe}(Q, m, \beta)$, and the definition of the lub.

4.2.5. Q.E.D.

PROOF: Steps 4.2.3 and 4.2.4 and the transitivity of $\sqsubseteq$ (assumption CS2) imply $w \sqsubseteq v$.

4.3. Q.E.D.

PROOF: By steps 4.1 and 4.2.

5. Q.E.D.

PROOF: Step 2 implies that steps 3 and 4 cover all possible cases.

Proposition 3 For any balnum $m > 0$, m-quorum Q, and ballot array β, if β is conservative then $\mathit{ProvedSafe}(Q, m, \beta)$ is non-empty.

PROOF: Let m be a balnum, Q an m-quorum, and β a conservative ballot array. Let k, $\mathcal{R}$, and γ be defined as in the definition of $\mathit{ProvedSafe}(Q, m, \beta)$. We assume that R_1 and R_2 are k-quorums in $\mathcal{R}$ (so $\mathcal{R}$ is non-empty) and show that $\gamma(R_1)$ and $\gamma(R_2)$ are compatible. The definition of *ProvedSafe* then implies that $\mathit{ProvedSafe}(Q, m, \beta)$ is non-empty.

1. CASE: k is a fast balnum.

1.1. Choose an acceptor a in $R_1 \cap R_2 \cap Q$

PROOF: a exists by the case 1 assumption that k is a fast balnum and the Quorum Assumption, since the definition of $\mathcal{R}$ implies that R_1 and R_2 are k-quorums.

1.2. Q.E.D.

PROOF: Step 1.1 and the definition of γ imply $\gamma(R_1) \sqsubseteq \beta_a[k]$ and $\gamma(R_2) \sqsubseteq \beta_a[k]$, so $\gamma(R_1)$ and $\gamma(R_2)$ are compatible.

2. CASE: k is a classic balnum.

2.1. Choose an upper bound w of $\{\beta_a[k] \mid (a \in Q) \land (\beta_a[k] \neq \mathit{none})\}$.

PROOF: w exists by case assumption 2 and the hypothesis that β is conservative.

2.2. Q.E.D.

PROOF: It follows from the definitions of $\mathcal{R}$ and γ that $\gamma(R) \sqsubseteq w$ for all R in $\mathcal{R}$. Hence R_1 and R_2 are compatible because they are in $\mathcal{R}$.

3. Q.E.D.

PROOF: By Steps 1 and 2 and the assumption that every balnum is either a fast or a classic one.

C TLA$^+$ Specifications

C.1 Command Structures

MODULE *OrderRelations*

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

CONSTANTS $S,\ \_ \preceq \_$

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

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

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

We now define the glb (greatest lower bound) and lub (least upper bound) operators. TLA$^+$ does not permit the use of $\sqcap$ and $\sqcup$ as prefix operators, so we use *GLB* and *LUB* for those operators. To define *GLB*, we first define $\mathit{IsLB}(lb, T)$ to be true iff lb is a lower bound of S, and $\mathit{IsGLB}(lb, T)$ to be true iff lb is a glb of S. The value of $\mathit{GLB}(T)$ is unspecified if T has no glb. The definitions for upper bounds are analogous.

$$\mathit{IsLB}(lb, T) \quad \triangleq \quad \land\ lb \in S$$

$$
\begin{array}{lll}
 & & \land \forall v \in T : lb \preceq v \\
IsGLB(lb, T) & \triangleq & \land IsLB(lb, T) \\
 & & \land \forall v \in S : IsLB(v, T) \Rightarrow (v \preceq lb) \\
GLB(T) & \triangleq & \text{CHOOSE } lb \in S : IsGLB(lb, T) \\
v \sqcap w & \triangleq & GLB(\{v, w\})
\end{array}
$$

$$
\begin{array}{lll}
IsUB(ub, T) & \triangleq & \land ub \in S \\
 & & \land \forall v \in T : v \preceq ub \\
IsLUB(ub, T) & \triangleq & \land IsUB(ub, T) \\
 & & \land \forall v \in S : IsUB(v, T) \Rightarrow (ub \preceq v) \\
LUB(T) & \triangleq & \text{CHOOSE } ub \in S : IsLUB(ub, T) \\
v \sqcup w & \triangleq & LUB(\{v, w\})
\end{array}
$$

MODULE *CStructs*

EXTENDS *Sequences*

The *Sequences* module defines the operator *Seq*.

We declare the assumed objects as parameters. TLA$^+$ does not permit the identifier $\perp$, so we use *Bottom* instead.

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

TLA$^+$ does not permit operator overloading, so we write $v ** \sigma$ instead of $v \bullet \sigma$ for a command sequence σ. TLA$^+$ allows recursive definitions only of functions, not operators, so the definition of $**$ recursively defines the function *conc* such that $conc[w, t] = w ** t$.

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

TLA$^+$ does not permit the general construct $\{e \mid P\}$, instead having two more restricted set-forming operators.

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

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. It is simpler to define the extended $\sqsubseteq$ relation here than to extend it later.

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

$v \sqsubset w \triangleq (v \sqsubseteq w) \land (v \neq w)$

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

$\textsc{instance}\ \mathit{OrderRelations}\ \textsc{with}\ S \leftarrow \mathit{CStruct},\ \preceq\ \leftarrow\ \sqsubseteq$

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

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

Here are the formal statements of assumptions CS1–CS4, as well as an assumption CS0 that was tacitly made but not explicitly named.

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

$$\mathit{CS1} \triangleq \mathit{CStruct} = \mathit{Str}(\mathit{Cmd})$$

$$\mathit{CS2} \triangleq \mathit{IsPartialOrder}$$

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

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

$\textsc{assume}\ \mathit{CS0} \land \mathit{CS1} \land \mathit{CS2} \land \mathit{CS3} \land \mathit{CS4}$

C.2 Generalized Consensus

MODULE *GeneralConsensus*

We specify the safety properties of the general consensus problem. We first give a "state-machine style" TLA^{+} specification *Spec*. We then assert that *Spec* implies the three safety properties Nontriviality, Stability, and Consistency.

EXTENDS *CStructs*

CONSTANT *Learner*

VARIABLES *propCmd*, *learned*

TypeInv asserts a type invariant; the assertion that *TypeInv* is always true is a property of (implied by) the specification.

$$\begin{aligned} TypeInv \triangleq\ & \land\ propCmd \subseteq Cmd \\ & \land\ learned \in [Learner \rightarrow CStruct] \end{aligned}$$

Init is the initial predicate.

$$\begin{aligned} Init \triangleq\ & \land\ propCmd = \{\} \\ & \land\ learned\ \ = [l \in Learner \mapsto Bottom] \end{aligned}$$

We now define the two actions of proposing a command and learning a c-struct. The *Learn* action sets *learned*$[l]$ to the lub of its present value and a proposed c-struct.

$$\begin{aligned} Propose \triangleq\ \exists\, C \in Cmd \setminus propCmd :\ & \land\ propCmd' = propCmd \cup \{C\} \\ & \land\ \text{UNCHANGED}\ learned \end{aligned}$$

$$\begin{aligned} Learn(l) \triangleq\ & \land\ \exists\, v \in Str(propCmd) : \\ & \qquad \land\ \forall\, r \in Learner : AreCompatible(v, learned[r]) \\ & \qquad \land\ learned' = [learned\ \text{EXCEPT}\ ![l] = learned[l] \sqcup v] \\ & \land\ \text{UNCHANGED}\ propCmd \end{aligned}$$

Next is the complete next-state action; *Spec* is the complete specification.

$$Next \triangleq Propose \lor \exists\, l \in Learner : Learn(l)$$

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

We now define the three safety properties as temporal formulas and assert that they and the type-correctness invariant are properties of the specification.

$$\begin{aligned} Nontriviality &\triangleq \forall\, l \in Learner : \Box(learned[l] \in Str(propCmd)) \\ Stability &\triangleq \forall\, l \in Learner,\ v \in CStruct : \\ & \qquad \Box((learned[l] = v) \Rightarrow \Box(v \sqsubseteq learned[l])) \\ Consistency &\triangleq \forall\, l1,\ l2 \in Learner : \\ & \qquad \Box AreCompatible(learned[l1],\ learned[l2]) \end{aligned}$$

THEOREM $Spec \Rightarrow (\Box\, TypeInv) \land Nontriviality \land Stability \land Consistency$

C.3 The Constant Operators of Paxos

MODULE *PaxosConstants*

This module defines the data structures for the abstract algoritm, introduced in Sections 5.1 and 5.2.

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 $\leq$ on ballot numbers is also a parameter.

CONSTANTS $BalNum$, $\_ \leq \_$, $IsFast(\_)$

We assume that 0 is a balnum, and that $\leq$ is a total ordering of the set *BalNum* of balnums. (Note: 0 is pre-defined in TLA$^+$ to have its usual value. However, this does not imply that *BalNum* contains any other usual numbers.)

ASSUME

$$\begin{array}{l} \land\ 0 \in BalNum \\ \land\ \text{LET } PO \triangleq \text{INSTANCE } OrderRelations \text{ WITH } S \leftarrow BalNum,\ \preceq\ \leftarrow\ \leq \\ \quad \text{IN } \quad PO!IsTotalOrder \end{array}$$

$$i < j \triangleq (i \leq j) \land (i \neq j)$$

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.

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

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

$$\begin{array}{l} QuorumAssumption \triangleq \\ \quad \land\ \forall m \in BalNum : Quorum(m) \subseteq \text{SUBSET } Acceptor \\ \quad \land\ \forall k, m \in BalNum : \\ \qquad \forall Q \in Quorum(k), R \in Quorum(m) : Q \cap R \neq \{\} \\ \quad \land\ \forall k \in BalNum : \\ \qquad IsFast(k) \Rightarrow \forall m \in BalNum : \\ \qquad\qquad \forall Q1, Q2 \in Quorum(k), R \in Quorum(m) : \\ \qquad\qquad\quad Q1 \cap Q2 \cap R \neq \{\} \end{array}$$

ASSUME $QuorumAssumption$

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 $\beta.vote[a][m]$ and $\widehat{\beta_a}$ as $\beta.mbal[a]$.

$$
\begin{array}{l}
\mathit{BallotArray} \triangleq \\
\quad \{\mathit{beta} \in [\mathit{vote} : [\mathit{Acceptor} \rightarrow [\mathit{BalNum} \rightarrow \mathit{CStruct} \cup \{\mathit{none}\}]], \\
\qquad\qquad \mathit{mbal} : [\mathit{Acceptor} \rightarrow \mathit{BalNum}]] : \\
\qquad \forall\, a \in \mathit{Acceptor} : \\
\qquad\quad \land\ \mathit{beta.vote}[a][0] \neq \mathit{none} \\
\qquad\quad \land\ \mathit{IsFiniteSet}(\{m \in \mathit{BalNum} : \mathit{beta.vote}[a][m] \neq \mathit{none}\}) \\
\qquad\quad \land\ \forall\, m \in \mathit{BalNum} : (\mathit{beta.mbal}[a] < m) \Rightarrow (\mathit{beta.vote}[a][m] = \mathit{none})\}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{IsChosenAt}(v,\ m,\ \mathit{beta}) \triangleq \\
\quad \exists\, Q \in \mathit{Quorum}(m) : \\
\qquad \forall\, a \in Q : (v \sqsubseteq \mathit{beta.vote}[a][m])
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{IsChoosableAt}(v,\ m,\ \mathit{beta}) \triangleq \\
\quad \exists\, Q \in \mathit{Quorum}(m) : \\
\qquad \forall\, a \in Q : (m < \mathit{beta.mbal}[a]) \Rightarrow (v \sqsubseteq \mathit{beta.vote}[a][m])
\end{array}
$$

$$
\begin{array}{l}
\mathit{IsSafeAt}(v,\ m,\ \mathit{beta}) \triangleq \\
\quad \forall\, k \in \mathit{BalNum} : \\
\qquad (k < m) \Rightarrow \forall\, w \in \mathit{CStruct} : \mathit{IsChoosableAt}(w,\ k,\ \mathit{beta}) \Rightarrow (w \sqsubseteq v)
\end{array}
$$

$$
\begin{array}{l}
\mathit{IsSafe}(\mathit{beta}) \triangleq \\
\quad \forall\, a \in \mathit{Acceptor},\ k \in \mathit{BalNum} : \\
\qquad (\mathit{beta.vote}[a][k] \neq \mathit{none}) \Rightarrow \mathit{IsSafeAt}(\mathit{beta.vote}[a][k],\ k,\ \mathit{beta})
\end{array}
$$

$$
\begin{array}{l}
\textsc{theorem}\quad \text{Proposition 1} \\
\quad \forall\, \mathit{beta} \in \mathit{BallotArray} : \\
\qquad \mathit{IsSafe}(\mathit{beta}) \Rightarrow \mathit{IsCompatible}(\{v \in \mathit{CStruct} : \mathit{IsChosenIn}(v,\ \mathit{beta})\})
\end{array}
$$

$$
\begin{array}{l}
\mathit{ProvedSafe}(Q,\ m,\ \mathit{beta}) \triangleq \\
\quad \textsc{let}\ k \quad \triangleq \mathit{Max}(\{i \in \mathit{BalNum} : \\
\qquad\qquad\qquad\qquad (i < m) \land (\exists\, a \in Q : \mathit{beta.vote}[a][i] \neq \mathit{none})\}) \\
\qquad\quad RS \quad \triangleq \{R \in \mathit{Quorum}(k) : \forall\, a \in Q \cap R : \mathit{beta.vote}[a][k] \neq \mathit{none}\} \\
\qquad\quad g(R) \triangleq \mathit{GLB}(\{\mathit{beta.vote}[a][k] : a \in Q \cap R\}) \\
\qquad\quad G \quad\ \triangleq \{g(R) : R \in RS\} \\
\quad \textsc{in}\quad \textsc{if}\ RS = \{\}\ \textsc{then}\ \{\mathit{beta.vote}[a][k] : \\
\qquad\qquad\qquad\qquad\qquad a \in \{b \in Q : \mathit{beta.vote}[b][k] \neq \mathit{none}\}\}
\end{array}
$$

$$\begin{array}{l} \text{ELSE IF } IsCompatible(G) \text{ THEN } \{LUB(G)\} \\ \qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } \{\} \end{array}$$

THEOREM Proposition 2

$$\begin{array}{l} \forall\, m \in BalNum \setminus \{0\},\ beta \in BallotArray : \\ \quad \forall\, Q \in Quorum(m) : \\ \qquad \land\ IsSafe(beta) \\ \qquad \land\ \forall\, a \in Q : m \leq beta.mbal[a] \\ \qquad \Rightarrow \forall\, v \in ProvedSafe(Q, m, beta) : IsSafeAt(v, m, beta) \end{array}$$

$$\begin{array}{l} IsConservative(beta) \triangleq \\ \quad \forall\, m \in BalNum,\ a,\ b \in Acceptor : \\ \qquad \land\ \neg IsFast(m) \\ \qquad \land\ beta.vote[a][m] \neq none \\ \qquad \land\ beta.vote[b][m] \neq none \\ \qquad \Rightarrow AreCompatible(beta.vote[a][m],\ beta.vote[b][m]) \end{array}$$

THEOREM Proposition 3

$$\begin{array}{l} \forall\, beta \in BallotArray : \\ \quad IsConservative(beta) \Rightarrow \\ \qquad \forall\, m \in BalNum \setminus \{0\} : \\ \qquad\quad \forall\, Q \in Quorum(m) : ProvedSafe(Q, m, beta) \neq \{\} \end{array}$$

C.4 The Abstract Algorithm

MODULE *AbstractGPaxos*

EXTENDS *PaxosConstants*

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

We begin with the type invariant and the initial predicate.

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

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

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \text{IF } m = 0 \text{ THEN } Bottom \text{ ELSE } none]],\\
\qquad\qquad mbal \mapsto [a \in Acceptor \mapsto 0]]\\
\quad \land\ maxTried = [m \in BalNum \mapsto\\
\qquad\qquad\qquad\quad \text{IF } m = 0 \text{ THEN } Bottom \text{ ELSE } none]\\
\quad \land\ minTried = maxTried
\end{array}
$$

We next define the three invariants of the abstract algorithm.

$$
\begin{array}{l}
TriedInvariant \triangleq\\
\quad \forall\, m \in BalNum : \land\ minTried[m] \sqsubseteq maxTried[m]\\
\qquad\qquad\qquad\qquad\ \land\ (minTried[m] \neq none) \Rightarrow\\
\qquad\qquad\qquad\qquad\qquad \land\ IsSafeAt(minTried[m], m, bA)\\
\qquad\qquad\qquad\qquad\qquad \land\ maxTried[m] \in Str(propCmd)
\end{array}
$$

$$
\begin{array}{l}
bAInvariant \triangleq\\
\quad \forall\, a \in Acceptor, m \in BalNum :\\
\qquad (bA.vote[a][m] \neq none) \Rightarrow\\
\qquad\quad \land\ minTried[m] \sqsubseteq bA.vote[a][m]\\
\qquad\quad \land\ \neg IsFast(m) \Rightarrow (bA.vote[a][m] \sqsubseteq maxTried[m])\\
\qquad\quad \land\ IsFast(m) \Rightarrow (bA.vote[a][m] \in Str(propCmd))
\end{array}
$$

$$
\begin{array}{l}
learnedInvariant \triangleq\\
\quad \forall\, l \in Learner : \land\ learned[l] \in Str(propCmd)\\
\qquad\qquad\qquad\quad\ \land\ \exists\, S \in \text{SUBSET } CStruct :\\
\qquad\qquad\qquad\qquad \land\ IsFiniteSet(S)\\
\qquad\qquad\qquad\qquad \land\ \forall\, v \in S : IsChosenIn(v, bA)\\
\qquad\qquad\qquad\qquad \land\ learned[l] = LUB(S)
\end{array}
$$

We now define the actions.

$$
\begin{array}{l}
Propose(C) \triangleq\\
\quad \land\ C \notin propCmd\\
\quad \land\ propCmd' = propCmd \cup \{C\}\\
\quad \land\ \text{UNCHANGED } \langle learned, bA, minTried, maxTried\rangle
\end{array}
$$

$$
\begin{array}{l}
JoinBallot(a, m) \triangleq\\
\quad \land\ bA.mbal[a] < m\\
\quad \land\ bA' = [bA \text{ EXCEPT } !.mbal[a] = m]\\
\quad \land\ \text{UNCHANGED } \langle propCmd, learned, minTried, maxTried\rangle
\end{array}
$$

$$
\begin{array}{l}
StartBallot(m, Q) \triangleq\\
\quad \land\ maxTried[m] = none\\
\quad \land\ \forall\, a \in Q : m \leq bA.mbal[a]
\end{array}
$$

$\land \exists\, w \in \mathit{ProvedSafe}(Q, m, bA), s \in \mathit{Seq}(\mathit{propCmd}) :$
$\quad \land\, \mathit{minTried}' = [\mathit{minTried} \text{ EXCEPT } ![m] = w ** s]$
$\quad \land\, \mathit{maxTried}' = [\mathit{maxTried} \text{ EXCEPT } ![m] = w ** s]$
$\land$ UNCHANGED $\langle \mathit{propCmd}, \mathit{learned}, bA \rangle$

$\mathit{Suggest}(m, C) \triangleq$
$\land\, C \in \mathit{propCmd}$
$\land\, \mathit{maxTried}[m] \neq \mathit{none}$
$\land\, \mathit{maxTried}' = [\mathit{maxTried} \text{ EXCEPT } ![m] = \mathit{maxTried}[m] \bullet C]$
$\land$ UNCHANGED $\langle \mathit{propCmd}, \mathit{learned}, bA, \mathit{minTried} \rangle$

$\mathit{ClassicVote}(a, v) \triangleq$
$\land\, \mathit{maxTried}[bA.\mathit{mbal}[a]] \neq \mathit{none}$
$\land\, \mathit{minTried}[bA.\mathit{mbal}[a]] \sqsubseteq v$
$\land\, v \sqsubseteq \mathit{maxTried}[bA.\mathit{mbal}[a]]$
$\land\, \lor\, bA.\mathit{vote}[a][bA.\mathit{mbal}[a]] = \mathit{none}$
$\quad \lor\, bA.\mathit{vote}[a][bA.\mathit{mbal}[a]] \sqsubset v$
$\land\, bA' = [bA \text{ EXCEPT } !.\mathit{vote}[a][bA.\mathit{mbal}[a]] = v]$
$\land$ UNCHANGED $\langle \mathit{propCmd}, \mathit{learned}, \mathit{minTried}, \mathit{maxTried} \rangle$

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

$\mathit{AbstractLearn}(l, v) \triangleq$
$\land\, \mathit{IsChosenIn}(v, bA)$
$\land\, \mathit{learned}' = [\mathit{learned} \text{ EXCEPT } ![l] = \mathit{learned}[l] \sqcup v]$
$\land$ UNCHANGED $\langle \mathit{propCmd}, bA, \mathit{minTried}, \mathit{maxTried} \rangle$

We combine the actions into the next-state relation and define *Spec* to be the complete specification.

$$
\begin{aligned}
\mathit{Next} \triangleq\ & \lor \exists\, C \in \mathit{Cmd} : \lor\, \mathit{Propose}(C) \\
& \qquad\qquad\qquad\ \lor \exists\, m \in \mathit{BalNum} : \mathit{Suggest}(m, C) \\
& \qquad\qquad\qquad\ \lor \exists\, a \in \mathit{Acceptor} : \mathit{FastVote}(a, C) \\
& \lor \exists\, m \in \mathit{BalNum} : \lor \exists\, a \in \mathit{Acceptor} : \mathit{JoinBallot}(a, m) \\
& \qquad\qquad\qquad\quad\ \lor \exists\, Q \in \mathit{Quorum}(m) : \mathit{StartBallot}(m, Q) \\
& \lor \exists\, v \in \mathit{CStruct} : \lor \exists\, a \in \mathit{Acceptor} : \mathit{ClassicVote}(a, v) \\
& \qquad\qquad\qquad\quad\ \lor \exists\, l \in \mathit{Learner} : \mathit{AbstractLearn}(l, v)
\end{aligned}
$$

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

The following theorem asserts the invariance of our invariants.

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

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

$GC \triangleq$ INSTANCE *GeneralConsensus*

THEOREM $Spec \Rightarrow GC!Spec$

C.5 The Distributed Abstract Algorithm

MODULE *DistAbstractGPaxos*

We import all the declarations and definitions from module *AbstractGPaxos*.

EXTENDS *AbstractGPaxos*

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.

$$\begin{aligned} Msg \triangleq\ & [type : \{\text{"propose"}\},\ cmd : Cmd] \\ \cup\ & [type : \{\text{"1a"}\},\ bal : BalNum] \\ \cup\ & [type : \{\text{"1b"}\},\ bal : BalNum,\ acc : Acceptor, \\ & \quad vote : [BalNum \to CStruct \cup \{none\}]] \\ \cup\ & [type : \{\text{"2a"}\},\ bal : BalNum,\ val : CStruct] \\ \cup\ & [type : \{\text{"2b"}\},\ bal : BalNum,\ acc : Acceptor,\ val : CStruct] \end{aligned}$$

We describe the state of the message-passing system by the value of the variable *msgs*. Because we are specifying only safety and not liveness, we do not need explicitly to model message loss. Since an action is never required to happen, the loss of a message during an execution of the system is modeled by the receive action for that message never being executed in the corresponding behavior. We can also model the possibility of receiving the same message multiple times by never deleting a message when it is received. So, we use a simple model of the message passing in which a message is sent by adding it to the set *msgs*, and a process can at any time receive any message that is an element of *msgs*.

VARIABLES *msgs*

We begin with the type invariant and the initial predicate. We prefix with a D standard names like *Init*, which are already defined in the *AbstractGPaxos*

$$\begin{aligned} DTypeInv \triangleq\ & \land\ TypeInv \\ & \land\ msgs \subseteq Msg \end{aligned}$$

$$DInit \triangleq\ \land\ Init$$

$\land\ msgs = \{\}$

We now define the actions. When an action very directly implements an action of the abstract non-distributed algorithm, we can re-use the action definition from module *AbstractGPaxos*.

$$
\begin{array}{l}
SendProposal(C) \triangleq \\
\quad \land\ propCmd' = propCmd \cup \{C\} \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“propose”},\ cmd \mapsto C]\} \\
\quad \land\ \text{UNCHANGED}\ \langle learned,\ bA,\ minTried,\ maxTried \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase1a(m) \triangleq \\
\quad \land\ maxTried[m] = none \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“1a”},\ bal \mapsto m]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ bA,\ minTried,\ maxTried \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase1b(a,\ m) \triangleq \\
\quad \land\ [type \mapsto \text{“1a”},\ bal \mapsto m] \in msgs \\
\quad \land\ JoinBallot(a,\ m) \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“1b”},\ bal \mapsto m,\ acc \mapsto a, \\
\qquad\qquad\qquad\qquad vote \mapsto bA.vote[a]]\}
\end{array}
$$

$$
\begin{array}{l}
Phase2Start(m,\ v) \triangleq \\
\quad \land\ maxTried[m] = none \\
\quad \land\ \exists\, Q \in Quorum(m) : \\
\qquad \land\ \forall\, a \in Q : \exists\, msg \in msgs : \begin{array}[t]{l} \land\ msg.type = \text{“1b”} \\ \land\ msg.bal \ \ = m \\ \land\ msg.acc \ \ = a \end{array} \\
\qquad \land\ \text{LET}\ beta \triangleq \\
\qquad\qquad\qquad \text{CHOOSE}\ b \in BallotArray : \\
\qquad\qquad\qquad\quad \forall\, a \in Q : \begin{array}[t]{l} \land\ b.mbal[a] = m \\ \land\ \exists\, msg \in msgs : \begin{array}[t]{l} \land\ msg.type = \text{“1b”} \\ \land\ msg.bal \ \ = m \\ \land\ msg.acc \ \ = a \\ \land\ b.vote[a] = msg.vote \end{array} \end{array} \\
\qquad\qquad\ \ pCmd \triangleq \\
\qquad\qquad\qquad \{msg.cmd : msg \in \{mg \in msgs : mg.type = \text{“propose”}\}\} \\
\qquad\quad \text{IN}\quad \exists\, w \in ProvedSafe(Q,\ m,\ beta),\ s \in Seq(pCmd) : \\
\qquad\qquad\qquad \land\ minTried' = [minTried\ \text{EXCEPT}\ ![m] = w ** s] \\
\qquad\qquad\qquad \land\ maxTried' = minTried' \\
\qquad\qquad\qquad \land\ msgs' = msgs \cup \{[type \mapsto \text{“2a”},\ bal \mapsto m,\ val \mapsto w ** s]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ bA \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase2aClassic(m,\ C)\ \triangleq \\
\quad \land\ [type \mapsto \text{“propose”},\ cmd \mapsto C] \in msgs \\
\quad \land\ maxTried[m] \neq none \\
\quad \land\ maxTried' = [maxTried\ \text{EXCEPT}\ ![m] = maxTried[m] \bullet C] \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“2a”},\ bal \mapsto m,\ val \mapsto maxTried'[m]]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ bA,\ minTried \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase2bClassic(a,\ m,\ v)\ \triangleq \\
\quad \land\ [type \mapsto \text{“2a”},\ bal \mapsto m,\ val \mapsto v] \in msgs \\
\quad \land\ bA.mbal[a] = m \\
\quad \land\ \lor\ bA.vote[a][bA.mbal[a]] = none \\
\quad \ \ \lor\ bA.vote[a][bA.mbal[a]] \sqsubset v \\
\quad \land\ bA' = [bA\ \text{EXCEPT}\ !.vote[a][bA.mbal[a]] = v] \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto m,\ acc \mapsto a,\ val \mapsto v]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ minTried,\ maxTried \rangle
\end{array}
$$

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

$$
\begin{array}{l}
Learn(l,\ v)\ \triangleq \\
\quad \land\ \ \exists\, m \in BalNum : \\
\qquad\quad \exists\, Q \in Quorum(m) : \\
\qquad\qquad \forall\, a \in Q : \exists\, msg \in msgs : \begin{array}[t]{l} \land\ msg.type = \text{“2b”} \\ \land\ msg.bal\ \ = m \\ \land\ msg.acc\ \ = a \\ \land\ v \sqsubseteq msg.val \end{array} \\
\quad \land\ \ learned' = [learned\ \text{EXCEPT}\ ![l] = learned[l] \sqcup v] \\
\quad \land\ \ \text{UNCHANGED}\ \langle propCmd,\ bA,\ minTried,\ maxTried,\ msgs \rangle
\end{array}
$$

DNext and *DSpec* are the complete next-state relation and specification.

$$
\begin{array}{l}
DNext\ \triangleq\ \lor\ \exists\, C \in Cmd : SendProposal(C) \quad \text{The proposers' actions.} \\
\qquad\qquad\ \lor\ \exists\, m \in BalNum : \quad \text{The leaders' actions.} \\
\qquad\qquad\qquad \lor\ Phase1a(m) \\
\qquad\qquad\qquad \lor\ \exists\, v \in CStruct : Phase2Start(m,\ v) \\
\qquad\qquad\qquad \lor\ \exists\, C \in Cmd : Phase2aClassic(m,\ C)
\end{array}
$$

$\vee \exists a \in Acceptor :$ The acceptors' actions.

$\quad \vee Phase1b(a, m)$

$\quad \vee \exists v \in CStruct : Phase2bClassic(a, m, v)$

$\quad \vee \exists C \in Cmd : Phase2bFast(a, m, C)$

$\vee \exists l \in Learner :$ The learners' actions.

$\quad \exists v \in CStruct : Learn(l, v)$

$DSpec \triangleq DInit \wedge \Box[DNext]_{\langle propCmd, learned, bA, minTried, maxTried, msgs \rangle}$

The following theorems assert that *DTypeInv* is an invariant and that *DSpec* implements the specification of generalized consensus, formula *Spec* of module *GeneralConsensus*.

THEOREM $DSpec \Rightarrow \Box DTypeInv$

THEOREM $DSpec \Rightarrow GC!Spec$

C.6 The Generalized Paxos Consensus Algorithm

MODULE *GeneralizedPaxos*

EXTENDS *PaxosConstants*

We introduce a set *Leader* of leaders, and let *LeaderOf*(m) be the leader of ballot number m.

CONSTANT *Leader*, *LeaderOf*$(\_)$

ASSUME $\forall m \in BalNum : LeaderOf(m) \in Leader$

The set *Msg* of all possible messages is the same as for the distributed abstract algorithm of module *DistAbstractGPaxos*.

$$
\begin{aligned}
Msg \triangleq\ & [type : \{\text{"propose"}\}, cmd : Cmd] \\
\cup\ & [type : \{\text{"1a"}\}, bal : BalNum] \\
\cup\ & [type : \{\text{"1b"}\}, bal : BalNum, acc : Acceptor, \\
& \quad vbal : BalNum, vote : CStruct \cup \{none\}] \\
\cup\ & [type : \{\text{"2a"}\}, bal : BalNum, val : CStruct] \\
\cup\ & [type : \{\text{"2b"}\}, bal : BalNum, acc : Acceptor, val : CStruct]
\end{aligned}
$$

We define *NotABalNum* to be an arbitrary value that is not a balnum.

$NotABalNum \triangleq \text{CHOOSE } m : m \notin BalNum$

The variables *propCmd*, *learned*, and *msgs* are the same as in the distributed abstract algorithm. We replace the abstract algorithm's variable *bA* with the variables *mbal*, *bal*, and *val*. The value of $curLdrBal[ldr]$ is the ballot that leader *ldr* is currently leading or has most recently led. Initially, its value is initially $curLdrBal[ldr]$ equals *NotABalNum* for all leaders except the leader of ballot 0, which is initially in progress. We replace the variable *maxTried* of the abstract algorithm with *maxLdrTried*, where the value of $maxLdrTried[ldr]$ corresponds to the value of $maxTried[curLdrBal[ldr]]$ in the abstract algorithm.

VARIABLES *propCmd*, *learned*, *msgs*, *maxLdrTried*, *curLdrBal*, *mbal*, *bal*, *val*

We begin with the type invariant and the initial predicate.

$$
\begin{aligned}
TypeInv \triangleq\ & \land propCmd \subseteq Cmd \\
& \land learned \in [Learner \rightarrow CStruct] \\
& \land msgs \subseteq Msg \\
& \land maxLdrTried \in [Leader \rightarrow CStruct \cup \{none\}] \\
& \land curLdrBal \in [Leader \rightarrow BalNum \cup \{NotABalNum\}] \\
& \land mbal \in [Acceptor \rightarrow BalNum] \\
& \land bal \in [Acceptor \rightarrow BalNum] \\
& \land val \in [Acceptor \rightarrow CStruct]
\end{aligned}
$$

$$
\begin{aligned}
Init \triangleq\ & \land propCmd = \{\} \\
& \land learned = [l \in Learner \mapsto Bottom] \\
& \land msgs = \{\} \\
& \land maxLdrTried = [ldr \in Leader \mapsto \textsc{if}\ ldr = LeaderOf(0) \\
& \qquad\qquad \textsc{then}\ Bottom\ \textsc{else}\ none] \\
& \land curLdrBal = [ldr \in Leader \mapsto \textsc{if}\ ldr = LeaderOf(0) \\
& \qquad\qquad \textsc{then}\ 0\ \textsc{else}\ NotABalNum] \\
& \land mbal = [a \in Acceptor \mapsto 0] \\
& \land bal = [a \in Acceptor \mapsto 0] \\
& \land val = [a \in Acceptor \mapsto Bottom]
\end{aligned}
$$

We now define the actions.

$$
\begin{aligned}
& SendProposal(C) \triangleq \\
& \quad \land propCmd' = propCmd \cup \{C\} \\
& \quad \land msgs' = msgs \cup \{[type \mapsto \text{“propose”}, cmd \mapsto C]\} \\
& \quad \land \textsc{unchanged}\ \langle learned, maxLdrTried, curLdrBal, mbal, bal, val \rangle
\end{aligned}
$$

$$
\begin{aligned}
& Phase1a(ldr) \triangleq \\
& \quad \land \exists\, m \in BalNum : \\
& \qquad \land \lor curLdrBal[ldr] = NotABalNum \\
& \qquad\quad \lor curLdrBal[ldr] < m \\
& \qquad \land LeaderOf(m) = ldr
\end{aligned}
$$

$$\begin{array}{l}
\quad\quad \land\ curLdrBal' \quad = [curLdrBal \quad \text{EXCEPT } ![ldr] \ = m] \\
\quad\quad \land\ maxLdrTried' = [maxLdrTried \text{ EXCEPT } ![ldr] = none] \\
\quad\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{"1a"},\ bal \mapsto m]\} \\
\quad \land \text{ UNCHANGED } \langle propCmd,\ learned,\ mbal,\ bal,\ val \rangle
\end{array}$$

$$\begin{array}{l}
Phase1b(a,\ m) \ \triangleq \\
\quad \land\ [type \mapsto \text{"1a"},\ bal \mapsto m] \in msgs \\
\quad \land\ mbal[a] < m \\
\quad \land\ mbal' = [mbal \text{ EXCEPT } ![a] = m] \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{"1b"},\ bal \mapsto m,\ acc \mapsto a, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad vbal \mapsto bal[a],\ vote \mapsto val[a]]\} \\
\quad \land \text{ UNCHANGED } \langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ bal,\ val \rangle
\end{array}$$

$$\begin{array}{l}
Phase2Start(ldr,\ v) \ \triangleq \\
\quad \land\ curLdrBal[ldr] \neq NotABalNum \\
\quad \land\ maxLdrTried[ldr] = none \\
\quad \land\ \exists\, Q \in Quorum(curLdrBal[ldr]) : \\
\quad\quad \land\ \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type = \text{"1b"} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ msg.bal \ \ = curLdrBal[ldr] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ msg.acc \ \ = a \\
\quad\quad \land \text{ LET}
\end{array}$$

We define $PrSafe$ so it equals the value of $ProvedSafe(Q, m, beta)$ where computed in the corresponding action of the distributed abstract algorithm. To help understand this correspondence, see the definition of $ProvedSafe$ in module $PaxosConstants$.

$$1bMsg(a) \ \triangleq$$

For an acceptor a in Q, this is the "1b" message sent by a for ballot number $curLdrBal[ldr]$. There can be only one such message.

$$\begin{array}{l}
\quad \text{CHOOSE } msg \in msgs : \ \land\ msg.type = \text{"1b"} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ msg.bal \ \ = curLdrBal[ldr] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ msg.acc \ \ = a \\
k \quad\ \ \triangleq\ Max(\{1bMsg(a).vbal : a \in Q\}) \\
RS \quad \triangleq\ \{R \in Quorum(k) : \forall\, a \in Q \cap R : 1bMsg(a).vbal = k\} \\
g(R)\ \triangleq\ GLB(\{1bMsg(a).vote : a \in Q \cap R\}) \\
G \quad\ \ \triangleq\ \{g(R) : R \in RS\} \\
PrSafe \ \triangleq
\end{array}$$

When the action is enabled, the set G will always be compatible.

$$\begin{array}{l}
\quad \text{IF } RS = \{\} \text{ THEN } \{1bMsg(a).vote : \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad a \in \{b \in Q : 1bMsg(b).vbal = k\}\} \\
\quad\quad\quad\quad\quad\quad \text{ELSE } \ \{LUB(G)\} \\
pCmd \ \triangleq\ \{msg.cmd : msg \in \{mg \in msgs : mg.type = \text{"propose"}\}\} \\
\text{IN } \quad \land\ \exists\, w \in PrSafe,\ s \in Seq(pCmd) :
\end{array}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \land\ maxLdrTried' = [maxLdrTried\ \text{EXCEPT}\ ![ldr] = w ** s] \\
\qquad\qquad\qquad\qquad \land\ msgs' = msgs \cup \{[type \mapsto \text{“2a”},\ bal \mapsto curLdrBal[ldr], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad val\ \ \mapsto w ** s]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ curLdrBal,\ mbal,\ bal,\ val \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase2aClassic(ldr,\ C)\ \triangleq \\
\quad \land\ curLdrBal[ldr] \neq NotABalNum \\
\quad \land\ [type \mapsto \text{“propose”},\ cmd \mapsto C] \in msgs \\
\quad \land\ maxLdrTried[ldr] \neq none \\
\quad \land\ maxLdrTried' = [maxLdrTried\ \text{EXCEPT}\ ![ldr] = maxLdrTried[ldr] \bullet C] \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“2a”},\ bal \mapsto curLdrBal[ldr], \\
\qquad\qquad\qquad\qquad\qquad val \mapsto maxLdrTried'[ldr]]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ curLdrBal,\ mbal,\ bal,\ val \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase2bClassic(a,\ v)\ \triangleq \\
\quad \land\ [type \mapsto \text{“2a”},\ bal \mapsto mbal[a],\ val \mapsto v] \in msgs \\
\quad \land\ \lor\ bal[a] < mbal[a] \\
\quad\quad\ \ \lor\ val[a] \sqsubset v \\
\quad \land\ bal'\ \ \ = [bal\ \text{EXCEPT}\ ![a] = mbal[a]] \\
\quad \land\ val'\ \ \ = [val\ \text{EXCEPT}\ ![a] = v] \\
\quad \land\ msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto mbal[a],\ acc \mapsto a,\ val \mapsto v]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ mbal \rangle
\end{array}
$$

$$
\begin{array}{l}
Phase2bFast(a,\ C)\ \triangleq \\
\quad \land\ [type \mapsto \text{“propose”},\ cmd \mapsto C] \in msgs \\
\quad \land\ bal[a] = mbal[a] \\
\quad \land\ val' = [val\ \text{EXCEPT}\ ![a] = val[a] \bullet C] \\
\quad \land\ msgs' = msgs\ \cup \\
\qquad\qquad\qquad \{[type \mapsto \text{“2b”},\ bal \mapsto mbal[a],\ acc \mapsto a,\ val \mapsto val'[a]]\} \\
\quad \land\ \text{UNCHANGED}\ \langle propCmd,\ learned,\ maxLdrTried,\ curLdrBal,\ mbal,\ bal \rangle
\end{array}
$$

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

Next and *Spec* are the complete next-state relation and specification.

$$
\begin{array}{l}
Next \triangleq \lor \exists C \in Cmd : SendProposal(C) \quad \text{The proposers' actions.} \\
\qquad \lor \exists ldr \in Leader : \quad \text{The leaders' actions.} \\
\qquad\qquad \lor Phase1a(ldr) \\
\qquad\qquad \lor \exists v \in CStruct : Phase2Start(ldr, v) \\
\qquad\qquad \lor \exists C \in Cmd : Phase2aClassic(ldr, C) \\
\qquad \lor \exists a \in Acceptor : \quad \text{The acceptors' actions.} \\
\qquad\qquad \lor \exists v \in CStruct : Phase2bClassic(a, v) \\
\qquad\qquad \lor \exists m \in BalNum : Phase1b(a, m) \\
\qquad\qquad \lor \exists C \in Cmd : Phase2bFast(a, C) \\
\qquad \lor \exists l \in Learner : \quad \text{The learners' actions.} \\
\qquad\qquad \exists v \in CStruct : Learn(l, v)
\end{array}
$$

$$
Spec \triangleq Init \land \Box[Next]_{\langle propCmd,\, learned,\, msgs,\, maxLdrTried,\, curLdrBal,\, mbal,\, bal,\, val \rangle}
$$

The following theorems assert that *TypeInv* is an invariant and that *Spec* implements the specification of generalized consensus, formula *Spec* of module *GeneralConsensus*.

THEOREM $Spec \Rightarrow \Box TypeInv$

$GC \triangleq$ INSTANCE *GeneralConsensus*

THEOREM $Spec \Rightarrow GC!Spec$