Byzantizing Paxos by Refinement

Leslie Lamport

Microsoft Research

28 June 2011
minor correction: 14 June 2012

Abstract

We derive a $3f+1$ process Byzantine Paxos consensus algorithm by Byzantizing a variant of the ordinary Paxos algorithm—that is, by having $2f+1$ nonfaulty processes emulate the ordinary Paxos algorithm despite the presence of f malicious processes. We have written a formal, machine-checked proof that the Byzantized algorithm implements the ordinary Paxos consensus algorithm under a suitable refinement mapping.

To appear in the Proceedings of the 25$^{\text{th}}$ International Symposium on Distributed Computing (DISC 2011)

Contents

You can verb anything.
Ron Ziegler (quoted by Brian Reid)

1 Introduction

The Paxos algorithm [6] has become a standard tool for implementing fault-tolerant distributed systems. It uses $2f + 1$ processes to tolerate the benign failure of any f of them. More recently, Castro and Liskov developed a $3f + 1$ process algorithm [2] that tolerates f Byzantine (maliciously faulty) processes. Intuitively, their algorithm seems to be a Byzantine version of Paxos. Other algorithms that also seem to be Byzantine versions of Paxos have subsequently appeared [4, 11, 14].

The only previous attempt we know of to explain the relation between a Byzantine Paxos algorithm and ordinary Paxos was by Lampson [13]. He derived both from an abstract, non-distributed algorithm. We take a more direct approach and derive a Byzantine Paxos algorithm from a distributed non-Byzantine one by a procedure we call *Byzantizing*, which converts an N process algorithm that tolerates the benign failure of up to f processes into an $N + f$ process algorithm that tolerates f Byzantine processes. In the Byzantized algorithm, the N good processes emulate the execution of the original algorithm despite the presence of f Byzantine ones. (Of course, a good process does not know which of the other processes are Byzantine.)

The heart of ordinary or Byzantine Paxos is a consensus algorithm. We Byzantize a variant of the classic Paxos consensus algorithm, which we call *PCon*, to obtain an abstract generalization of the Castro-Liskov Byzantine consensus algorithm that we call *BPCon*. (Section 3 explains why we do not Byzantize the original Paxos consensus algorithm.)

It is easy to make something appear simple by hand-waving. The fact that *BPCon* is derived from *PCon* is expressed formally by a TLA$^+$ [7] theorem asserting that *BPCon* implements *PCon* under a suitable refinement mapping [1]. (A derivation is an implementation proof presented backwards.) A formal proof of the safety part of this theorem has been written and checked by the TLAPS proof system; it is available on the Web [5]. We discuss liveness informally. We believe that other Byzantine Paxos consensus algorithms can also be derived by Byzantizing versions of Paxos, but we have not proved any other derivation.

We describe algorithms *PCon* and *BPCon* informally here. Their formal specifications are on the Web, along with the correctness proof [5]. (A pretty-printed version of the algorithms' PlusCal [9] code is also available on the

Web site.) Section 8 explains just what this proof proves. In Section 7, we describe how the Castro-Liskov algorithm refines algorithm *BPCon*.

2 Consensus and Classic Paxos

We assume the usual distributed-computing model of asynchronous processes communicating by messages. By a benign failure, we mean the loss of a message or a process stopping. A Byzantine process may send any message, but we assume that the identity of the sender of a message can be determined by the receiver. This can be achieved by either point-to-point communication or message authenticators (MACs), which are described in Section 6.1.

2.1 Consensus

In a complete specification of consensus, *proposer* processes propose values, a set of *acceptor* processes together choose one of the proposed values, and *learner* processes learn what value, if any, has been chosen. The algorithm must tolerate the failure of some number f of acceptor processes, as well as the failure of any proposer or learner process.

To simplify the formal development, we eliminate the proposers and learners, and we consider only acceptors. Our definition of what value is chosen makes it clear how learning is implemented. Implementing proposers is not trivial in the Byzantine case, since one must prevent a Byzantine process from pretending to be a nonfaulty proposer. It becomes trivial by using digital signatures, and Castro and Liskov explain how it is done with MACs.

With this simplification, the specification of consensus consists of a trivial algorithm in which the acceptors can choose at most one value, but once chosen a value must remain forever chosen.

It is well-known that fault-tolerant consensus cannot be implemented in a purely asynchronous system [3]. We require that the safety properties (at most one value chosen and a value never unchosen) hold even in the absence of any synchrony assumption, and that liveness (a value is eventually chosen) holds under suitable synchrony assumptions on nonfaulty processes and the communication among them.

2.2 Paxos Consensus

The classic Paxos consensus algorithm was described in [6] and independently stated without proof by Oki [15]. It performs numbered ballots, each orchestrated by a leader. Multiple ballots may be performed concurrently (with different leaders). Once an acceptor performs an action in a ballot, it never performs any further actions of a lower-numbered ballot. We assume that ballots are numbered by natural numbers.

Let N be the number of acceptors, where $N > f$, and let a *quorum* be any $N-f$ acceptors. For safety, we require that any two quorums have a non-empty intersection, which is true if $N > 2f$. The only other property of quorums we use is that there is a quorum consisting entirely of nonfaulty processes, which is required for liveness.

An acceptor can *vote* for at most one value in any ballot. A value v is *chosen in* a ballot iff a quorum of acceptors have voted for v in that ballot. A value is *chosen* iff it is chosen in some ballot.

We say that a value v is *safe at* a ballot number b if no value other than v has been chosen or ever can be chosen in any ballot numbered less than b. (Although described intuitively in temporal terms, *safe at* is actually a function of the algorithm's current state.) The algorithm maintains the following properties:

P1. An acceptor can vote for a value v in ballot b only if v is safe at b.

P2. Different acceptors cannot vote for different values in the same ballot.

These properties are maintained by having the ballot-b leader choose a single value v that is safe at b and asking the acceptors to vote for v in ballot b. An acceptor will vote only when it receives such a request (and only if it has not performed any action of a higher-numbered ballot). A ballot b proceeds in two phases, with the following actions.

Phase 1a The ballot-b leader sends a $1a$ message to the acceptors.

Phase 1b An acceptor responds to the leader's ballot-b $1a$ message with a $1b$ message containing the number of the highest-numbered ballot in which it has voted and the value it voted for in that ballot, or saying that it has cast no votes.

Phase 2a Using the $1b$ messages sent by a quorum of acceptors, the leader chooses a value v that is safe at b and sends a $2a$ message containing v to the acceptors.

Phase 2b Upon receipt of the leader's ballot-b $2a$ message, an acceptor votes for v in ballot b by sending a $2b$ message.

(Remember that an acceptor performs a ballot-b Phase 1b or 2b action only if it has not performed an action for a higher-numbered ballot.) A value v is *chosen* iff a quorum of acceptors have voted for v in some ballot. A learner learns that a value has been chosen if it receives $2b$ messages from a quorum of acceptors for the same ballot (which by P2 must all report votes for the same value). However, since we are not modeling learners, the $2b$ messages serve only to record votes.

In its Phase 2a action, the ballot-b leader must determine a safe value from the ballot-b $1b$ messages it receives from a quorum. It does this by using the following properties of the algorithm.

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b.

P3b. If some acceptor in the quorum has voted, let c be the highest-numbered ballot less than b in which such a vote was cast. The value voted for in ballot c is safe at b. (By P2, there is only one such value.)

Paxos implements a state machine by executing an infinite sequence of separate instances of the consensus algorithm. There is normally a single leader executing ballots, using the same ballot number in all the instances. If that leader fails, a new leader executes Phase 1 for a higher-numbered ballot simultaneously for all instances of the consensus algorithm. For all instances in which a ballot was begun but learners may not know the chosen value, Phase 2 is executed immediately. For ballots not begun, in which P3a holds, the leader waits until it receives the necessary client proposals before executing Phase 2.

The ballot-b leader can always execute the Phase 1a action, and it can execute the Phase 2a action if it has received $1b$ messages from a quorum of acceptors. An acceptor can respond to messages from the leader if it has received no message from a higher-numbered ballot. Therefore, the ballot-b leader and a nonfaulty quorum of acceptors can choose a value if no higher-numbered ballot is begun. The liveness property satisfied by classic Paxos consensus is obtained directly from this observation; we will not bother stating it precisely. We just point out that the essential property from which liveness follows is the ability of the ballot-b leader to determine a safe value in Phase 2a from the ballot-b $1b$ messages sent by a quorum of acceptors.

3 Byzantizing An Algorithm

We Byzantize a consensus algorithm by having N acceptors emulate it in the presence of f *fake* acceptors—Byzantine processes that pretend to be acceptors. (Everything works with $m \leq f$ fake acceptors, but for simplicity we omit this generalization.) We sometimes call the acceptors *real* to more clearly distinguish them from the fake acceptors. Processes other than acceptors may be Byzantine—in particular, a Byzantized Paxos algorithm must tolerate malicious leaders. However, assumptions about the non-malicious behavior of leaders is required for liveness.

Formally, emulation means performing an action that, under a refinement mapping, is an action of the emulated algorithm. A refinement mapping maps each state of the emulating system (the implementation) to a state of the emulated one (the specification). Refinement mappings are explained in more detail in Section 8.

We are effectively assuming that which processes may be malicious are determined in advance. Since the Byzantized algorithm assumes no knowledge of which are the real acceptors and which the fake ones, this assumption results in no loss of generality. (It can be viewed as adding a prophecy variable [1] whose value always equals the set of processes that may fail.) Moreover, since a malicious process can do anything, including acting like a nonfaulty one, we can prove that the algorithm tolerates at least f malicious acceptors by assuming that there are exactly f fake acceptors that are malicious from the start.

We define the set of *byzacceptors* to be the union of the sets of real and fake acceptors. We define a *byzquorum* to be a set of byzacceptors that is guaranteed to contain a quorum of acceptors. If a quorum consists of any q acceptors, then a byzquorum consists of any $q+f$ byzacceptors. For liveness, we need the assumption that the set of all real acceptors (which we assume never fail) form a byzquorum.

In the Byzantized algorithm, a nonfaulty process must ensure that each action in its emulation is enabled by the original algorithm. For example, if we were modeling learners, the action of learning that a value v is chosen would be enabled by the receipt of ballot-b $2b$ messages with value v from a quorum of acceptors. In the Byzantized algorithm, the learner could perform that action when it had received such messages from a byzquorum, since that set of messages would contain a subset from a quorum of acceptors.

The key action in Paxos consensus is the leader's Phase 2a action, which chooses a safe value based on properties P3a and P3b. The leader can deduce that P3a holds if it receives $1b$ messages from a byzquorum, each asserting

that the sender has not voted, because that byzquorum contains a quorum of acceptors. However, P3b is problematic. In the original algorithm, it is satisfied if there is a $1b$ message from some single acceptor reporting a vote in a ballot c. However, in the Byzantized algorithm, there is no way to determine if a single message is from a real or fake acceptor. One can maintain safety by requiring that a vote be reported in the highest-numbered ballot c by $f+1$ byzacceptors. However, liveness would then be lost because it is possible to reach a state in which this condition does not hold for the $1b$ messages sent by the real acceptors.

One way to fix this problem is to assume $N > 3f$. In that case, any two quorums have at least $f+1$ acceptors in common, and we can replace P3a and P3b by

P3a$'$. If there is no ballot numbered less than b in which $f+1$ acceptors have voted, then all values are safe at b.

P3b$'$. If there is some ballot c in which acceptors have voted and there is no higher-numbered ballot less than b in which $f+1$ acceptors have voted, then the value v voted for in c is safe at b.

The Phase 2a action is then always enabled by the receipt of $1b$ messages from a byzquorum because, if P3a$'$ does not hold, then we can apply P3b$'$ with c the largest ballot in which $f+1$ byzacceptors have voted for the same value. However, this is unsatisfactory because if assumes $N > 3f$, so it leads to a Byzantine consensus algorithm requiring more than $4f$ acceptors. Our solution to this problem is to use the variant of the Paxos consensus algorithm described in Section 4 below.

There is still another problem to be solved. For a Phase 2a action to be enabled, it is not enough for the leader to have received $1b$ messages from a quorum; it is also necessary that the leader has not already sent a (different) $2a$ message. If P3a holds, a malicious leader could send two $2a$ messages for different safe values. This could lead to two different values being chosen in two later ballots.

The solution to this problem lies in having the leader and the acceptors cooperatively emulate the execution of the Phase 2a action, using a new Phase 2av action. The leader sends to the byzacceptors a request to execute the Phase 2a action for a particular value v. An acceptor responds to this request by executing a Phase 2av action in which it sends a $2av$ message with value v to all the byzacceptors. It executes the Phase 2av action only if (i) it can determine that one such $2a$ message could be sent in the emulated algorithm (we explain in Section 5 how it does this), and (ii) it has not

already executed a Phase 2av action in the current ballot. An acceptor can execute the Phase 2b action if it has received $2av$ messages with the same value from a byzquorum. Since any two byzquorums have a (real) acceptor in common, no two acceptors can execute Phase 2b actions for different values. The refinement mapping is defined so an emulated $2a$ message is considered to have been sent when a quorum of acceptors have sent the corresponding $2av$ messages.

4 Algorithm *PCon*

We now describe a variant called *PCon* of the classic Paxos consensus algorithm. As explained below, a more general version of this algorithm has appeared before. Like classic Paxos, it assumes N acceptors with $N > 2f+1$.

In the classic algorithm described above, a ballot-b $2a$ message serves two functions: (i) it asserts that a value is safe at b, and (ii) it instructs the acceptors to vote for that value in ballot b. In algorithm *PCon*, we introduce a $1c$ message to accomplish (i), and we allow the leader to send multiple $1c$ messages asserting that multiple values are safe. We introduce a leader Phase 1c action and modify the Phase 2a action as follows:

Phase 1c Using the $1b$ messages from a quorum of acceptors, the leader chooses a set of values that are safe at b and sends a $1c$ message for each of those values.

Phase 2a The leader sends a $2a$ message for some value for which it has sent a $1c$ message.

The leader does not have to send all its $1c$ messages at once; it can execute the Phase 1c action multiple times in a single ballot. To choose safe values in the Phase 1c action, the ballot-b leader uses the following properties of the algorithm after receiving $1b$ messages from a quorum of acceptors.

P3a. If no acceptor in the quorum has voted in a ballot numbered less than b, then all values are safe at b.

P3c. If a ballot-c $1c$ message with value v has been sent, for some $c < b$, and (i) no acceptor in the quorum has voted in any ballot greater than c and less than b, and (ii) any acceptor in the quorum that has voted in ballot c voted for v in that ballot, then v is safe at b.

The careful reader will have noticed that we have not specified to whom the ballot-b leader sends its $1c$ messages, or how it learns about $1c$ messages sent in lower-numbered ballots so it can check if P3c holds. In algorithm *PCon*, the $1c$ messages are logical constructs that need not actually be sent. Sending a $2a$ message implies that the necessary $1c$ message was sent, and a $1b$ message reporting a vote in ballot c implies that a ballot-c $1c$ message was sent. So, why were $1c$ messages introduced in previous algorithms?

Systems that run for a long time cannot be based on a fixed set of acceptors. Acceptors must occasionally be removed and new ones added—a procedure called *reconfiguration*. In classic Paxos, reconfiguration happens between consensus instances, and a single instance is effectively executed by a single set of acceptors. Two algorithms have been proposed in which reconfiguration happens within the execution of a single consensus instance, with different ballots using possibly different sets of acceptors: Vertical Paxos [10] and an unpublished version of Cheap Paxos [12]. The $1c$ messages serve to eliminate the dependence on acceptors from lower-numbered ballots, which may have been reconfigured out of the system. When a new active leader begins ballot b, case P3a holds for the infinitely many instances for which Phase 2 of ballot b has not yet begun. The leader's $1c$ messages inform future leaders of this fact, so they do not have to learn about votes cast in any ballot numbered less than b.

The astute reader will have observed that the definition of *safe at* implies that if two different values are safe at b, then all values are safe at b. There is no reason for the leader to do anything other than sending a message saying a single value is safe, or sending messages saying that all values are safe. However, the more general algorithm is just as easy to prove correct and is simpler to Byzantize.

5 Algorithm *BPCon*

We now derive algorithm *BPCon* by Byzantizing the N-acceptor algorithm *PCon*, adding f fake acceptors. We first consider the actions of a leader process. There is no explicit $2a$ message or Phase 2a action in algorithm *BPCon*. Instead, the acceptors cooperate to emulate the sending of a $2a$ message, as described above in Section 3. The ballot-b leader requests that a Phase 2a action be performed for a value v for which it has already sent a $1c$ message. On receiving the first such request, an acceptor executes a Phase 2av action, sending a ballot-b $2av$ message for value v, if it has already received a legal ballot-b $1c$ message with that value.

Since the leader's request is necessary only for liveness, we do not explicitly model it. Instead, we allow an acceptor to perform a ballot-b Phase 2av action iff it has received the necessary $1c$ action and has not already sent a ballot-b $2av$ message.

Because the algorithm must tolerate malicious leaders, we let the ballot-b leader send any $1a$ and $1c$ messages it wants. (Remember that we assume a process cannot send a message that appears to be from another process.) There is only one possible ballot-b $1a$ message, and algorithm *PCon*'s Phase 1a action allows the leader to send it at any time. Hence the *BPCon* Phase 1a action is the same as the corresponding *PCon* action. The *BPCon* Phase 1c action allows the ballot-b leader to send any ballot-b $1c$ message at any time.

Acceptors will ignore a $1c$ message unless it is legal. To ensure liveness, a nonfaulty leader must send a message that (real) acceptors act upon. To see how it does that, we must determine how an acceptor knows that a $1c$ message is legal.

The sending of a ballot-b $1c$ message is enabled in *PCon* by P3a or P3c above, which requires the receipt of a set of $1b$ messages from a quorum and possibly of a $1c$ message. In *BPCon*, we put into the $1b$ messages additional information to enable the deduction that a $1c$ message was sent. An acceptor includes in its $1b$ messages the set of all $2av$ messages that it has sent—except that for each value v, it includes (and remembers) only the $2av$ message with the highest numbered ballot that it sent for v. Each of those $2av$ messages was sent in response to a legal $1c$ message. As explained in our discussion of Byzantizing in Section 3, this implies that given a set S of ballot-b $1b$ messages sent by a byzquorum, the following two conditions imply P3a and P3c, respectively:

BP3a. Each message in S asserts that its sender has not voted.

BP3c. For some $c < b$ and some value v, (a) each message in S asserts that (i) its sender has not voted in any ballot greater than c and (ii) if it voted in c then that vote was for v, and (b) there are $f+1$ $1b$ messages (not necessarily in S) from byzacceptors saying that they sent a $2av$ message with value v in ballot c.

A little thought shows we can weaken condition (b) of BP3c to assert:

> (b′) there are $f+1$ $1b$ messages from byzacceptors saying that they sent a $2av$ message with value v in a ballot $\geq c$.

The c of P3c is then the largest of those ballot numbers $\geq c$ reported by a real acceptor.

To determine if a $1c$ message is legal, each acceptor maintains a set of $1b$ messages that it knows have been sent. Our abstract algorithm assumes an action that nondeterministically adds to that set any subset of $1b$ messages that have actually been sent. Of course, some of those $1b$ messages may be from fake acceptors, which may send any $1b$ message. Liveness requires the leader to ensure that the acceptors eventually know that the $1b$ messages enabling its sending of the $1c$ message have been sent. We discuss in Section 6 below how that is done.

As described above in Section 3, an acceptor performs a $Phase2b$ action when it knows that it has received identical $2av$ messages from a quorum of acceptors. A $2a$ message of *PCon* is emulated by a set of identical $2av$ messages sent by a quorum, with the Phase 2a action emulated by the sending of the last of that set of messages.

6 Liveness and Learning About Sent Messages

Liveness of *PCon* requires that a nonfaulty leader executes a ballot b, no leader begins a higher-numbered ballot, and the leader and nonfaulty acceptors can communicate with one another. The requirements for liveness of *BPCon* are the same. However, it is difficult to ensure that a Byzantine leader does not execute a higher-numbered ballot. Doing this seems to require an engineering solution based on real-time assumptions. One such solution is presented by Castro and Liskov.

Assuming these requirements, liveness of *BPCon* requires satisfying the following two conditions:

BL1. The leader can find $1b$ messages satisfying $BP3a$ or $BP3c$.

BL2. All real acceptors will know that those messages have been sent.

These two conditions imply that the leader will send a legal $1c$ message, a byzquorum BQ of real (nonfaulty) acceptors will receive that $1c$ message and send $2av$ messages, all the acceptors in BQ will receive those $2av$ messages and send $2b$ messages. Learners, upon receiving those $2b$ messages will learn that the value has been chosen.

To show that BL1 holds, observe that the ballot-b leader will eventually receive $1b$ messages from the acceptors in BQ. Let S be the set of those $1b$ messages. We now show that BP3a or BP3c holds.

1. It suffices to assume that BP3a is false and prove BP3c.

   Proof Obvious.

2. Let c be the largest ballot in which an acceptor in BQ voted, let a be such an acceptor, and let v be the value it voted for.

   Proof The existence of such a c follows from 1.

3. Acceptor a received ballot-c $2av$ messages with value v from a byzquorum.

   Proof By 2 and the enabling condition of the Phase 2b action.

4. No acceptor voted for a value other than v in ballot c.

   Proof By 3, since any two byzquorums have an acceptor in common and an acceptor can send at most one ballot-c av message.

5. At least $f+1$ acceptors sent ballot-c $2av$ messages with value v.

   Proof By 3, since a byzquorum contains at least $f+1$ acceptors.

6. Condition (b$'$) of BP3c holds.

   Proof By 5, because an acceptor sending a ballot-c $2av$ message with value v implies that, for $b > c$, its ballot-b $1b$ message will report that it sent a $2av$ message with value v in some ballot $\geq c$.

7. Condition (a) of BP3c holds.

   Proof By 2 (no acceptor in BQ voted in a ballot $> c$) and 4.

8. QED

   Proof By 1, 6, and 7.

This shows that BL1 eventually holds. To prove liveness, we need to show that BL2 holds. To ensure that it holds, the leader must have a way of ensuring that all the real acceptors eventually learn that a $1b$ message was sent. If the $1b$ message was sent by a real acceptor, then that acceptor can just broadcast its $1b$ message to all the byzacceptors as well as to the leader. We now present two methods for ensuring that an acceptor learns that a $1b$ message was sent, even if it was sent by a fake acceptor.

6.1 Sending Proofs

The simplest approach is for the leader to include with its $1c$ message a proof that all the necessary $1b$ messages have been sent. The easiest way to do that is to use full digital signatures and have byzacceptors sign their $1b$ messages. The leader can just include the necessary properly signed $1b$ messages in its $1c$ message.

There is another way for the leader to include in its $1c$ message a proof that a message was sent, using only authentication with MACs. A MAC is a signature $m_{p \to q}$ that a process p can attach to a message m that proves to q that p sent m. The MAC $m_{p \to q}$ proves nothing to any process other than q. We now describe a general method of obtaining a proof of a fact in the presence of f Byzantine processes. We can apply it to the fact that a process p sent a particular message.

Suppose a message m asserts a certain fact, and process q receives it with MAC $m_{p \to q}$ from $f+1$ different processes p. With at most f Byzantine processes, at least one of those processes p asserting the fact is nonfaulty, so the fact must be true. However, q cannot prove to any other process that the fact is true. However, suppose that it receives from $2f+1$ processes p the message together with a vector $\langle m_{p \to r_1}, \ldots, m_{p \to r_k} \rangle$ of MACs for the k processes $r_1, \ldots, r_k$. At least $f+1$ of those vectors were sent by nonfaulty processes p, so they have correct MACs and will therefore convince each r_i that a nonfaulty process sent m. Therefore, q can send m and these $2f+1$ vectors of MACs to each of the processes r_i as a proof of the fact asserted by m.

In general, a vector of $(j+1)f + 1$ MACs provides a proof that can be sent along a path of length j. For *BPCon*, we need it only for $j = 1$; one method of Byzantizing fast Paxos [8] uses the $j = 2$ case [11].

6.2 Relaying $1b$ Messages

We now describe another way a leader can ensure that good acceptors learn that a $1b$ message was sent. We have byzacceptors broadcast their $1b$ messages to all byzacceptors (as well as to the leader), and have them relay the $1b$ messages to the leader and to all other byzacceptors. Upon receipt of copies of a $1b$ message from $2f+1$ byzacceptors, the leader knows that at least $f+1$ real acceptors sent or relayed that message to all byzacceptors. Assuming the requirements for liveness, this implies that all acceptors will eventually receive copies of the $1b$ message from $f+1$ different byzacceptors, from which they infer that the message actually was sent.

This is the basic method used by Castro and Liskov. However, in their algorithm, the byzacceptors relay the broadcast $1b$ messages (which they call *view-change-acks*) only to the leader (which they call the *primary*). The leader includes (digests) of the $1b$ messages in its $1c$ message, and an acceptor asks the other byzacceptors to relay any $1b$ message that it hasn't received that is in the $1c$ message.

7 The Castro-Liskov Algorithm

The Castro-Liskov algorithm, like Paxos, executes a state machine by executing an unbounded sequence of instances of a consensus algorithm. It contains engineering optimizations for dealing with the sequence of instances—in particular, for garbage collecting old instances and for transferring state to repaired processes. We believe that those optimizations can be obtained by Byzantizing the corresponding optimizations for classic Paxos, but they are irrelevant to consensus. Some other optimizations, such as sending message digests instead of messages, are straightforward details that we ignore for simplicity.

When we ignore these details and consider only the consensus algorithm at the heart of the Castro-Liskov algorithm, we are left with an algorithm that refines *BPCon*. In the Castro-Liskov algorithm, byzacceptors are called *replicas*. The ballot-b leader is the replica called the *primary*, other byzacceptors being called *backups*. The replicas also serve as learners.

We explain how the Castro-Liskov consensus algorithm refines *BPCon* by describing how the messages of *BPCon* are implemented. We assume the reader is familiar with their algorithm.

- $1a$ There is no explicit $1a$ message; its sending is emulated cooperatively by the replicas when they decide to begin a view change.
- $1b$ This is the *view-change* message.
- $1c$ During a view change, the *new-view* message acts like $1c$ messages for all the consensus instances. For an instance in which the primary instructs the replicas to choose a specific value, it is a $1c$ message with that value. For all other instances, it is a set of $1c$ messages for all values. (Condition BP3a holds in those other instances.) The acceptors check the validity of these $1c$ messages simultaneously for all instances.

$2av$ This is a backup's *prepare* message. The *pre-prepare* message of the primary serves as its $2av$ message and as the message (not modeled in *BPCon*) that requests a Phase 2a action.

$2b$ This is the *commit* message.

As explained in Section 6.2, the Castro-Liskov algorithm's *view-change-ack* is used to relay $1b$ messages to the leader. Its *reply* message is sent by replicas serving as learners to inform the client of the chosen value.

We explained in Section 3 the difficulty in Byzantizing classic Paxos. Our inability to obtain the Castro-Liskov algorithm from classic Paxos is not a deficiency of Byzantizing; it is due to the fact that the algorithm does not refine classic Paxos—at least, not under any simple refinement mapping. In the Castro-Liskov consensus algorithm, a leader may be required to pre-prepare a value v even though no replica ever committed v in a previous view. This cannot happen in classic Paxos.

8 The Formal Specifications and Proof

All of our specifications and proofs are available on the Web [5]. Here, we give an overview of what we have done.

In addition to deriving *BPCon* from *PCon*, we also derive *PCon* as follows. We start with a simple specification *Consensus* of consensus. We refine *Consensus* by an algorithm *Voting*, a high-level non-distributed consensus algorithm that describes the ballots and voting that underlie *PCon*. We then obtain *PCon* by refining *Voting*.

All the specifications are written in PlusCal, a high-level algorithm language that superficially resembles a toy programming language [9]. A PlusCal algorithm is automatically translated into a TLA$^+$ specification. It is these TLA$^+$ specifications that we verify. The specifications of *BPCon* and *PCon* describe only the safety properties of the algorithms, so we are verifying only safety for them. For *Voting* and *Consensus*, we have written specifications of both safety and liveness.

Each step in the derivation of *BPCon* from *Consensus* is described formally by a refinement mapping. To explain what this means, we review refinement mappings as defined in [1]. (They are slightly different in TLA$^+$, which uses a single state space for all specifications.)

Let Σ_S denote the state space of a specification S, and let Σ_S^ω be the set of sequences of states in Σ_S. The specification S is a predicate on Σ_S^ω,

where $S(\sigma)$ is true for a state sequence σ iff σ represents a behavior (possible execution) permitted by S.

A refinement of a specification S by a specification R is described by a mapping ϕ from Σ_R to Σ_S. We extend ϕ in the obvious way (pointwise) to a mapping from Σ_R^ω to Σ_S^ω. If F is a function on Σ_S or Σ_S^ω, we define $\overline{F}$ to be the function on Σ_R or Σ_R^ω, respectively, that equals $F \circ \phi$. We say that R *refines* (or *implements*) S under ϕ iff R implies $\overline{S}$. Thus, verifying correctness of the refinement means verifying the formula $R \Rightarrow \overline{S}$.

We have proved the correctness of the refinement of *PCon* by *BPCon*, and our proof has been completely checked by the TLAPS proof system, with two exceptions:

- A few trivial facts about finite sets are assumed without proof—for example, that a finite, non-empty set of integers contains a maximal element. We used TLC to check for errors in the TLA$^+$ formulas that state these assumptions.

- A handful of simple steps in the complete TLA$^+$ proof require temporal-logic reasoning. These steps, and their proofs, are identical for every TLA$^+$ refinement proof of safety properties. Since TLAPS does not yet handle temporal reasoning, proofs of these steps were omitted.

We have also written a complete proof that *Voting* refines *Consensus*, including the liveness properties. Most of the non-temporal steps of that proof have been checked by TLAPS; see [5] for details. We have checked our refinement of *Voting* by *PCon* with the TLC model checker, using a large enough model to be confident that there are no "coding" errors in our specifications. That, combined with our understanding of the algorithms, gives us confidence that this refinement is correct.

Mathematical correctness of a refinement tells us nothing useful unless the refinement mapping is a useful one. For example, a simple counter implements *PCon* under a refinement mapping in which the counter changing from n to $n + 1$ is mapped to the execution of the Phase 1a action by the ballot-n leader. Our refinement mappings are intuitively reasonable, as indicated by our informal description of the refinement mapping for the refinement of *PCon* by *BPCon*. Because the purpose of a consensus algorithm is to determine what value is chosen, we can provide the following more rigorous demonstration that our refinement mappings are the "right" ones.

For each of our specifications, we define a state function *chosen* that equals the set of values that have been chosen (a set containing at most one

value). Let $chosen_S$ be the state function *chosen* defined for specification S. The following relations among these state functions show that our refinement mappings are the ones we want.

$$\begin{array}{lcl} chosen_{Voting} & = & \overline{chosen_{Consensus}} \\ chosen_{PCon} & = & \overline{chosen_{Voting}} \\ chosen_{BPCon} & \Rightarrow & \overline{chosen_{PCon}} \end{array}$$

The last relation is implication rather than equality for the following reason. A value v is in $\overline{chosen_{PCon}}$ iff a quorum of acceptors have voted for v in the same ballot. It is impossible for a learner to determine if that is true because it does not know which byzacceptors are acceptors. We therefore define $chosen_{BPCon}$ to contain a value v iff a byzquorum has voted for it in the same ballot, which implies that v is in $\overline{chosen_{PCon}}$; hence the implication. (For liveness, we must show that any element of $\overline{chosen_{PCon}}$ is eventually an element of $chosen_{BPCon}$).

The first of these relations is essentially the definition of the refinement mapping under which *Voting* refines *Consensus*. The second has been checked by TLC. The third has been proved and the proof checked by TLAPS.

9 Conclusion

For a number of years, we have been informally explaining Byzantine consensus algorithms as the Byzantizing of ordinary Paxos. We decided that formalizing the Byzantizing of Paxos would be interesting in itself and would provide a test of how well TLAPS works on real problems.

Although the basic idea of Byzantizing was right, the formalization revealed that we were quite wrong in the details. In particular, we originally thought that the Castro-Liskov algorithm refined classic Paxos consensus. We wrote and checked almost the complete proof of that refinement, discovering the error only because we were unable to prove one of the last remaining steps. We are not sure if we would have found the error had we written a careful, hierarchically structured hand proof. We are quite sure that we would not have found it by writing a conventional paragraph-style "mathematical" proof.

Our proof that *BPCon* refines *PCon* revealed a number of problems with TLAPS that were then corrected. Our subsequent proof that *Voting* refines *Consensus* went quite smoothly. We intend to finish checking that proof when TLAPS supports temporal reasoning. We hope to prove that

BPCon also refines *PCon* when suitable liveness properties are added to the specifications. It would be nice to prove that *PCon* refines *Voting*. However, we probably won't bother because we are already convinced that the refinement is correct, and because its simpler proof would be less of a test of TLAPS.

References

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

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[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] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

[5] Leslie Lamport. Mechanically checked safety proof of a byzantine paxos algorithm. URL `http://research.microsoft.com/users/lamport/tla/byzpaxos.html`. The page can also be found by searching the Web for the 23-letter string obtained by removing the "`-`" from `uid-lamportbyzpaxosproof`.

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

[7] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003.

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

[9] Leslie Lamport. The pluscal algorithm language. In *Theoretical Aspects of Computing—ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer-Verlag, 2009.

[10] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In Srikanta Tirthapura and Lorenzo Alvisi,

editors, *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009*, pages 312–313. ACM, 2009.

[11] Leslie B. Lamport. Fast byzantine paxos. United States Patent 7620680, filed August 15, 2002, issued November 17, 2009.

[12] Leslie B. Lamport and Michael T. Massa. Cheap Paxos. United States Patent 7249280, filed June 18, 2004, issued July 24, 2007.

[13] Butler W. Lampson. The ABCDs of Paxos. `http://research.microsoft.com/lampson/65-ABCDPaxos/Abstract.html`.

[14] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, pages 402–411, Yokohama, June 2006. IEEE Computer Society.

[15] Brian Masao Oki. Viewstamped replication for highly available distributed systems. Technical Report MIT/LCS/TR-423, MIT Laboratory for Computer Science, August 1988. (Ph.D. Thesis).