Fast Byzantine Paxos

Jean-Philippe Martin, Lorenzo Alvisi
Department of Computer Sciences
University of Texas at Austin
Email: {jpmartin, lorenzo}@cs.utexas.edu

1 Introduction

The consensus problem can be described in terms of the actions taken by three classes of agents: *proposers*, who propose values, *acceptors*, who together are responsible for choosing a single proposed value, and *learners*, who must learn the chosen value [10]. A single processor can act as more than one kind of agent. Consensus can be specified using the following three safety properties and two liveness properties:

CS1 Only a value that has been proposed may be chosen.

CS2 Only a single value may be chosen.

CS3 Only a chosen value may be learned by a correct learner.

CL1 Some proposed value is eventually chosen.

CL2 Once a value is chosen, correct learners eventually learn it.

Since the unearthing of the simple and practical Paxos protocol [13], consensus, which for years had largely been the focus of theoretical papers, has once again become popular with practitioners. This popularity should not be surprising, given that consensus is at the core of the state machine approach [9, 18], the most general method for implementing fault tolerant services in distributed systems. Yet, many practitioners had been discouraged by the provable impossibility of solving consensus deterministically in asynchronous systems with one faulty process [6]. Paxos offers the next best thing: while it cannot guarantee progress in some scenarios, it always preserves the safety properties of consensus, despite asynchrony and process crashes. More specifically, in Paxos one of the proposers is elected leader and it communicates with the acceptors. Paxos guarantees progress only when the leader is unique and can communicate with sufficiently many acceptors, but it ensures safety even with no leader or with multiple leaders.

Paxos is also attractive because it can be made very efficient in *gracious executions*, i.e. executions where there is a unique correct leader, all correct acceptors agree on its identity, and all correct replicas and all links between them are timely. Except in pathological situations, it is reasonable to expect that gracious executions will be the norm, and so it is desirable to optimize for them. For instance, FastPaxos [1] in a gracious execution requires only two communication steps[1] to reach consensus in non-Byzantine environments, matching the lower bound formalized by Keidar and Rajsbaum [7]. Consequently, in a state machine that uses FastPaxos, once the leader receives a client request it takes just two communication steps, in the common case, before the request can be executed. Henceforth, we use the terms "common case" and "gracious execution" interchangeably.

In this paper, we too focus on improving the common case performance of Paxos, but in the Byzantine model. Recent work has shown how to extend the Paxos consensus protocol to support Byzantine fault tolerant state machine replication. The resulting systems perform surprisingly well: they add modest latency [2], can proac-

[1] To be precise, this bound is only met for *stable intervals* in which no replica transitions between the crashed and "up" state.

tively recover from faults [3], can make use of existing software diversity to exploit opportunistic N-version programming [17], and can be engineered to protect confidentiality and reduce the replication costs incurred to tolerate f faulty state machine replicas [19].

Byzantine Paxos protocols fall short of the original, however, in the number of communication steps required to reach consensus in the common case. After a client request has been received by the leader, Byzantine protocols need a minimum of three additional communication steps (rather than the two required in the non-Byzantine case) before the request can be executed.

The main contribution of this paper is to present Fast Byzantine (or *FaB*) Paxos, the first Byzantine Paxos protocol, as far as we know, that requires only two communication steps to reach consensus in the common case.

Confirming a conjecture by Lamport [12], we find that the reduced latency comes at a price: FaB Paxos requires $5f + 1$ acceptors to tolerate f Byzantine acceptors, instead of the $3f + 1$ needed by previous protocols. For traditional implementations of the state machine approach, in which the roles of proposers, acceptors and learners are performed by the same set of machines, the extra replication required by our protocol may appear prohibitively large, especially when considering the software costs of implementing N-version programming (or opportunistic N-version programming) to eliminate correlated Byzantine faults [17].

We submit, however, that the new architecture for Byzantine fault tolerant state machine replication that we have recently introduced [19] makes this tradeoff look much more attractive. The key principle of this new architecture is to physically separate agreement from execution. In our architecture, a cluster of acceptors or *agreement replicas* is responsible for producing a linearizable order of client requests, while a separate cluster of learners or *execution replicas* executes the ordered requests.

For the purposes of this paper, the key advantage of this new architecture is that decoupling agreement from execution leads to agreement replicas (i.e. our acceptors) that are much simpler and less expensive than state machine replicas used in traditional architectures—and can therefore be more liberally used. In particular, our agreement replicas are cheaper both in terms of hardware—because of reduced processing, storage, and I/O requirements—and, especially, in terms of software: application-independent agreement replicas can be engineered as a generic library that may be reused across applications, while with traditional replicas the costs of N-version programming must be paid anew with each different service.

This paper is organized as follows. After discussing related work and system model in Sections 2 and 3, we present in Section 4 our main contribution—a Byzantine Paxos protocol that in the common case terminates in two communication steps. In the following section we show how to use FaB Paxos to achieve fast Byzantine fault-tolerant state machine replication, and finally we draw our conclusions.

2 Related Work

Consensus and state machine replication have generated a gold mine of papers. The veins from which our work derives are mainly those that originate with Lamport's Paxos protocol [13] and Castro and Liskov's work on Practical Byzantine Fault-tolerance (PBFT) [2]. In addition, the techniques we use to reduce the number of communication steps are inspired by the work on Byzantine quorum systems pioneered by Malkhi and Reiter [14].

The two protocols that are closest to FaB Paxos are the FastPaxos protocol by Boichat and colleagues [1], and Kursawe's Optimistic asynchronous Byzantine agreement [8]. Both protocols share our basic goal: to optimize the performance of the consensus protocol when runs are, informally speaking, well-behaved. Just like FaB Paxos, when runs are well-behaved both protocols can (i) operate without expensive public key cryptography operations and (ii) reach consensus in only two communication steps.

The differences arise in characterizing precisely what qualifies as good behavior and what is the extent of bad behavior that these protocols are designed to tolerate.

The conditions under which FastPaxos achieves consensus in two communication steps are quite similar to those needed by FaB Paxos to be equally fast. FastPaxos can deliver consensus in two communication steps during *stable periods*, i.e. periods where no process crashes or recovers, a majority of processes are up, and correct processes agree on the identity of the leader. The conditions under which we achieve gracious executions are somewhat weaker than these, in that during gracious executions processes can fail, provided that the leader does not fail. Another difference is that FastPaxos does not rely on eventual synchrony but on an eventual leader oracle. Since we only use the eventual synchrony assumption to provide leader election, our protocol also applies to the eventual leader oracle model. The most significant difference between the two protocols lies in the failure model they support: in FastPaxos processes can only fail by crashing, while in FaB Paxos they can fail arbitrarily. However, FastPaxos only requires $2f + 1$ acceptors, compared to the $5f + 1$ used in FaB Paxos.

In contrast to FastPaxos, Kursawe's elegant optimistic protocol assumes the same Byzantine failure model that we adopt and operates with only $3f + 1$ acceptors, instead of $5f + 1$.

However, the notion of what makes an execution well-behaved is much stronger for Kursawe's protocol than for FaB Paxos. In particular, his optimistic protocol achieves consensus in two communication steps only as long as channels are timely and *no* process is faulty: a single faulty process causes the fast optimistic agreement protocol to be permanently replaced by a traditional pessimistic, and slower, implementation of agreement. To be fast, FaB Paxos only requires gracious executions, which are compatible with process failures as long as there is a unique correct leader and all correct acceptors agree on its identity.

In his paper on lower bounds for asynchronous consensus' [11], Lamport, in his "approximate theorem" 3a, conjectures a bound $N > 2Q + F + 2M$ on the minimum number N of acceptors required by 2-step Byzantine consensus, where: (i) F is the maximum number of acceptor failures despite which consensus liveness is ensured; (ii) M is the maximum number of acceptor failures despite which consensus safety is ensured; and (iii) Q is the maximum number of acceptor failures despite which consensus must be 2-step. Lamport 's conjecture is more general than ours—we do not distinguish between M, F, and Q—and more restrictive—Lamport does not allow Byzantine learners; we do. Lamport's claim does not technically hold in the corner case where no learner can fail (see Appendix C). Dutta, Guerraoui and Vukolić have recently derived a proof of Lamport's original claim under the implicit assumption that at least one learner may fail [4].

3 System Model

We make no assumption about the relative speed of processes or communication links, or about the existence of synchronized clocks. The network is unreliable: messages can be dropped, reordered, inserted or duplicated. However, if a message is sent infinitely many times then it arrives at its destination infinitely many times. Finally, the recipient of a message knows who the sender is. In other words, we are using authenticated asynchronous fair links.

Following Paxos [10], we describe the behavior of FaB Paxos in terms of the actions performed by three classes of agents: proposers, acceptors, and learners. We assume $3f + 1$ proposers, $5f + 1$ acceptors, and $3f + 1$ learners. Note that a single physical replica may play multiple roles in the protocol. Each class contains at most f Byzantine faulty agents. When we consider FaB Paxos in connection with state machine replication, we assume that an arbitrary number of clients of the state machine can be Byzantine.

4 Fast Byzantine Consensus

We build FaB Paxos in stages: we start by describing a simple version of the protocol that relies on relatively strong assumptions and we proceed by progressively weakening the assumptions and refining the protocol accordingly. The complete FaB Paxos protocol that we obtain by the end of this section provides the following guarantees for each instance c of consensus:

- Safety (CS1-CS3).

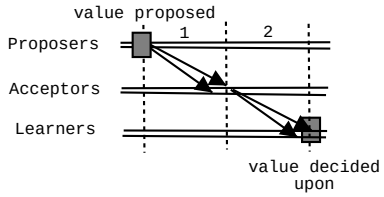

Figure 1: Common case operation

- Liveness (CL1-CL2) if the common case assumptions hold during c.
- Low latency (consensus in two communication steps) in the common case.

A *timely* link delivers all messages between correct nodes within a finite (but unspecified) time bound. Similarly, a timely replica processes all messages within a finite unspecified time bound.

In the state machine section we examine running consensus repeatedly. In that case, if the common case eventually holds, then eventually all instances of consensus terminate in two communication steps.

4.1 FaB Paxos in the Common Case

We first describe how FaB Paxos works during gracious executions when there is a unique correct leader and all correct acceptors agree on its identity. Links between correct replicas, as well as the correct replicas themselves are timely. At most f proposers are malicious (but not the leader), at most f acceptors are malicious, and at most f learners are malicious.

In this model, we can satisfy safety (CS1-CS3) and liveness (CL1-CL2) in two communication steps (see Figure 1).

In the first step, the leader sends its proposal to the acceptors. In the second step, the acceptors forward their value to the learners. Each learner then learns which value was chosen.

Acceptors only accept the first value that is sent to them. We say that a value v is *chosen* if and only if at least $3f + 1$ correct acceptors accept v. We will use this definition to prove that FaB Paxos satisfies CS1, CS2, and CL1.

Correct learners *learn* v once they see that at least $4f + 1$ acceptors accepted it.

The correctness proofs are simple, in part because of the strong assumptions we are making in this section. The safety proofs, however, hold without modification for much weaker assumptions, as we will see later.

4.1.1 Correctness in the common case

Lemma 1 (CS1). *Only a value that has been proposed may be chosen.*

Proof. Correct acceptors only accept values that are sent by a proposer. □

Lemma 2 (CS2). *Only a single value may be chosen.*

Proof. There are at most $5f+1$ correct acceptors (fewer than two groups of $3f + 1$) and correct acceptors accept a single value. □

Lemma 3 (CS3). *Only a chosen value may be learned by a correct learner.*

Proof. By definition, a correct learner learns v only if it sees that at least $4f + 1$ acceptors accepted v. Since at most f acceptors are faulty, then at least $3f + 1$ correct acceptors have accepted v. Then, by definition, v was chosen. □

Lemma 4 (CL1). *A proposed value is eventually chosen.*

Proof. The leader sends its proposal v to all acceptors. Since links are reliable and the leader is correct, all correct acceptors receive v. Since there are at least $4f + 1$ correct acceptors, v is chosen. □

Lemma 5 (CL2). *Once a value is chosen, correct learners can eventually learn it.*

Proof. If value v is chosen then, by definition, $3f + 1$ correct acceptors have accepted v. Since the leader is correct, it must have sent the same v to all acceptors. Further, since links are reliable, eventually all $4f+1$ correct acceptors will receive v. Upon receiving v, each correct acceptor sends v to all learners. Since links are reliable, all correct learners will eventually receive v from $4f + 1$ acceptors and learn v. □

4.2 Fair Links and Retransmission

So far we have assumed timely links. While this is a reasonable assumption in the common case, our protocol must also be able to handle periods of asynchrony. In this section we weaken our network model to consider fair asynchronous authenticated links (see Section 3).

We refine the protocol presented in Section 4.1 so that it still satisfies both safety (CS1-CS3) and liveness (CL1-CL2) under the weaker network model. Note, however, that now consensus may take more than two communication steps to complete, e.g. when all messages sent by the leader in the first step are dropped.

We use an end-to-end retransmission strategy: The leader sends its proposal to all acceptors repeatedly until it gets an ACK from $2f+1$ learners (out of $3f+1$).

Acceptors forward their accepted value to the learners every time they receive a proposed value from the leader and, as in the common case protocol, correct acceptors accept at most one value. When they learn a value and in response to every subsequent acceptor message, learners send an ACK to the leader.

Consensus requires all correct learners to learn the same value. However, the above "push" retransmission protocol leaves open the possibility that ACKs from faulty learners will cause the leader to stop retransmitting before all learners get a chance to learn the proposed value. To counter this problem, we supplement retransmission with a "pull" protocol: learners periodically query their peers and learn value v if $f+1$ of their peers learned v. Note that these exchanges are not necessary when no message is dropped.

4.2.1 Correctness of retransmission

The safety proofs are identical to those in the previous section. The liveness proofs are different, because we cannot rely on reliable links anymore, but only on fair links. Before showing that retransmission will allow the protocol to eventually complete, we show that faulty nodes cannot defeat our push protocol.

Lemma 6. *Faulty nodes cannot cause a correct leader to stop resending before some value is learned by at least $f+1$ correct learners.*

Proof. A correct leader resends until it receives $2f+1$ ACK from learners. Correct learners only send ACK messages after learning a value. If fewer than $f+1$ correct learners learned a value, then faulty learners cannot cause the leader to stop resending because there are only f of them. Since faulty acceptors and proposers have no impact on retransmission, the lemma immediately follows. □

Lemma 7 (CL1). *A proposed value is eventually chosen.*

Proof. Since no value can be learned by a correct learner before being chosen (CS3), Lemma 6 implies that the correct leader will resend at least until a value is chosen.

The correct leader repeatedly sends its proposal to all acceptors. Since links are fair, eventually all correct acceptors receive the leader's proposal. Since there are more than $3f+1$ correct acceptors, the proposed value is eventually chosen. □

Lemma 8 (CL2). *Once a value is chosen, correct learners can eventually learn it.*

Proof. Lemma 6 shows that the leader will not stop resending until $f+1$ correct learners acknowledge. This eventually happens: after a value v is chosen (Lemma 7), the acceptors repeatedly send that value to all learners. Since links are fair, eventually at least $f+1$ correct learners will have learned v.

The pull protocol ensures that all the other correct learners also eventually learn v: They periodically query their peers and learn v when they see that $f+1$ other learners have learned v. □

4.3 Failure Detector and Leader Election

So far, we have assumed that all replicas agree on a correct leader from the start. We now show how proposers can detect faulty leaders and elect a replacement. We delay a proof of the correctness of the resulting protocol until the next section, where we present the recovery protocol used by FaB Paxos when a faulty leader is detected.

We are now operating under the weakest model for which FaB Paxos is designed, in which as many as f proposers, f acceptors and f learners can be

Byzantinely faulty. We assume that at the beginning of the run of FaB Paxos links are fair, asynchronous and authenticated. There exists however some time T after which the system enters a *stable* phase during which both correct replicas and the links between them are timely. This model was introduced by Dwork [5].

4.3.1 Failure detector

The failure detector is based on a classical timeout mechanism. Because of space limitations we list the properties of the failure detector below. We refer the reader to Appendix A.1 for the proofs and a more detailed explanation.

Lemma 9. *A leader that makes no progress is eventually suspected by all correct proposers.*

Lemma 10. *Eventually, no correct proposer suspects a correct leader.*

4.3.2 Leader election

Our leader election protocol follows closely the one used by Castro and Liskov in PBFT [2]. Despite the few minor changes that we introduce (e.g. we use neither checkpoints certificates nor prepared certificates) the proof of correctness of the PBFT leader election protocol applies to ours as well.

4.4 Recovery Protocol

When proposers suspect the current leader of being faulty, they elect a new leader who then invokes the recovery protocol. There are two scenarios that require special care.

First, some value v may have already been chosen: the new leader must then propose the same v to maintain property CS2.

Second, a previous malicious leader may have performed a *poisonous write* [16], i.e. a write that prevents learners from reading any value—for example, a malicious leader could write a different value to each acceptor. If the new leader is correct, consensus should nonetheless terminate.

In the protocols discussed in Sections 4.1 and 4.2, the first scenario was addressed by requiring acceptors to accept a single value. Unfortunately, enforcing this requirement in the second scenario would prevent termination. Hence, in the full version of FaB Paxos we allow acceptors to change their values in response to a new proposal. Naturally, we must take precautions to ensure that CS2 still holds. In particular, our recovery protocol ensures the following:

1. If the new leader l is correct and a value v had been chosen prior to l's election, then l will propose v.
2. Eventually either a value is chosen or another leader is elected to replace l.

4.4.1 Introducing change vouchers

To regulate the conditions under which a correct acceptor accepts multiple values, we introduce the notion of a *change voucher*. Intuitively, we would like a change voucher to help ensure CS2 by providing correct acceptors with reliable information about the progress of the protocol. A change voucher should let a correct acceptor a determine whether a value has already been chosen, and, if so, should vouch for that value. A leader that wants a to accept a new value accompanies its request with a change voucher; a complies with the change when either the change voucher establishes that no value has been chosen or if it vouches for the value proposed by the leader.

Unfortunately, a faulty new leader could now use a change voucher *twice* to cause two different values to be chosen. Further, this can happen even if individual proposers only accept a given change voucher once. Consider the following situation. We split the acceptors into four groups. The first group has size $2f+1$, the second has size f and contains malicious acceptors, and the third and fourth have size f. Suppose the values they have initially accepted are "A","B","B", and "C", respectively. A malicious new leader l can gather a change voucher establishing that no value has been chosen With this voucher, l can first sway f acceptors from the third group to "A" (by definition, "A" is now chosen) , and then, using the same change voucher, persuade the acceptors in the first and fourth group to change their value to "B"—"B" is now chosen. Clearly, this execution violates CS2.

4.4.2 Implementation of change vouchers

In order to prevent faulty leaders from using change vouchers more than once, we make two

changes to our protocol.

First, we require each proposed value to be associated with a unique proposal sequence number (psn): a proposal now consists of a pair (v, psn). Every time a correct leader proposes a new value, it increments the proposal sequence number. Correct acceptors accept v only if they receive (v, psn) and one of the following conditions holds: (i) they have not accepted any previous proposal; (ii) they last accepted ov on receiving the pair $(ov, opsn)$, and $ov = v$ and $opsn \leq psn$; or (iii) they last accepted ov on receiving $(ov, opsn)$, the new proposal comes with a change voucher $(cv, cpsn)$ such that $opsn \leq cpsn$ and $cpsn = psn - 1$, and the change voucher either fails or vouches for v.

Note that now change vouchers vouch not just for a value v, but also for the associated proposal sequence number.

Second, we change the definition of *chosen*: a value v is chosen if there are $3f+1$ correct acceptors who at some point in time accept the *same pair* (v, psn). Similarly, learners now only learn v if they receive the same pair (v, psn) from $4f+1$ different acceptors.

The new leader gathers the change voucher by asking a quorum of $4f + 1$ responsive acceptors for the last value pair they have accepted (or $\perp$ if they have not accepted anything yet). Acceptors digitally sign their replies. When the leader has gathered $4f + 1$ replies with the same proposal sequence number, the set of replies gathered by the new leader constitutes a *change voucher*. If the change voucher contains some value $v \neq \perp$ at least $2f+1$ times, then we say that it *vouches* for v. Otherwise we say that it *fails*.

As we prove in the next section, change vouchers have the following three properties.

1. A change voucher can vouch for at most one value.

2. If some value (v, psn) is chosen, then all change vouchers $(cv, cpsn)$ with $cpsn \geq psn$ vouch for v.

3. Conversely, if a change voucher $(cv, cpsn)$ fails, then no value (v, psn) with $psn \leq cpsn$ is ever chosen.

If the change voucher vouches for some value v, then the new leader l resumes the normal leader protocol using v as the value to be proposed. Otherwise, l resumes the normal leader protocol using a value of its choosing. In both cases, l piggybacks the change voucher alongside its proposal to justify its choice of value.

Let us revisit the troublesome scenario of before in light of these changes. Suppose, without loss of generality, that the malicious leader l gathers a failed change voucher for proposal sequence number 0. To have "A" chosen, l performs two steps: first, l sends a new proposal ("A", 1) to the acceptors in the first group; then l sends ("A", 1) together with the failed change voucher for proposal 0 to the acceptors in the third group. Note that the first step is critical to have "A" chosen, as it ensures that the $3f + 1$ correct acceptors in the first and third group accept the same (v, psn) pair.

Fortunately, this first step is also what prevents l from using the failed change voucher to sway the acceptors in the first group to accept "B". Because they have last accepted the pair ("A", 1), when l presents the acceptors in the first group the failed change voucher for proposal sequence number 0, they will refuse it as too low, on account of (iii).

4.4.3 Correctness of change vouchers

Change vouchers contain $4f + 1$ values with the same proposal sequence number. When the new leader starts gathering replies for the change voucher, different acceptors might have different proposal sequence numbers. The gathering phase eventually terminates because there is no danger in advancing a given acceptor to a higher *psn* without modifying the proposal's value. The new leader therefore estimates the maximal proposal sequence number *max-psn* (using the largest *psn* it has seen so far), tells the acceptors to advance to it, and then attempts to gather the change voucher. If *max-psn* is not large enough (because the new leader has not heard from some acceptor with a larger proposal number), then the leader adjusts *max-psn* and tries again. Since this adjustment is only necessary the first time the leader hears from a given acceptor, the loop is guaranteed to terminate.

We now prove the three properties of change vouchers that we claimed in the previous section. The first property (a change voucher can vouch

for at most one value) trivially derives from the size of change vouchers, and the third property is equivalent to the second one. Remains to prove the second property.

Lemma 11. *If some value (v, psn) is chosen, then all change vouchers $(cv, cpsn)$ with $cpsn \geq psn$ vouch for v.*

Proof. By contradiction. Let $cpsn \geq psn$ be smallest proposal sequence number of any change voucher that does not vouch for v. That change voucher was assembled by the leader from the replies of a set X of $4f + 1$ acceptors. By definition, if (v, psn) is chosen then there is exists a set V of $3f + 1$ correct acceptors that all accept (v, psn) at some point in time. Since there are $5f + 1$ servers in total, V and X intersect in a set Y of at least $2f + 1$ correct acceptors.

We now narrow down the time at which acceptors in Y must have accepted (v, psn). They cannot accept (v, psn) after responding to the gather message with a value different from v, for after the gather message they can only change their value by incrementing their proposal sequence number to $cpsn+1$, which is strictly greater than psn. So all acceptors in Y either (1) already have value v when they reply to the gather message and later accept (v, psn), or (2) they accept (v, psn), then accept a different value, and then reply to the gather message. If all acceptors in Y fell under option (1), then the change voucher would have vouched for v. So there must be some acceptor s who accepts (v, psn) and then accepts a different value before replying to the gather message.

By rule (iii), for acceptor s to change its value from (v, psn) to some other value $(v', psn'), v' \neq v$, s must have accepted a change voucher $(cv', cpsn')$ vouching for v' with $psn \leq cpsn'$ and $cpsn' + 1 = psn'$. Since s is in X, s's reply was taken into account in the change voucher that had $cpsn$ as its proposal sequence number; thus $psn' \leq cpsn$. Putting it all together yields $cpsn' < cpsn$. This violates our hypothesis that $cpsn$ is the smallest proposal sequence number among the change vouchers that do not vouch for v. □

4.4.4 Correctness with the leader election and recovery protocols

The proofs for CS1, CS3 and CL2 are unchanged.

Lemma 12 (CS2). *Only a single value may be chosen.*

Proof. By contradiction. Suppose that two values (v, psn) and (v', psn') are both chosen, and $v \neq v'$. Since there are $5f + 1$ acceptors in total (more than two groups of $3f + 1$), there is at least one correct acceptor s who accepted (say) (v, psn) first, and (v', psn') second. Since $v \neq v'$, s must have accepted a change voucher $(cv, cpsn)$ that either fails or vouches for v', with $psn \leq cpsn < psn'$. By lemma 11, $(cv, cpsn)$ must vouch for v since (v, psn) was chosen and $psn \leq cpsn$. □

Lemma 13 (CL1). *Some proposed value is eventually chosen.*

Proof. Since only a value that has been proposed may be chosen (CS1), it is sufficient to show that some value is eventually chosen.

We first show that if the proposal retransmission protocol stops, then a value has been chosen. Leaders only stop retransmitting in three cases: (i) they get an acknowledgment from at least one correct learner, (ii) they are faulty and choose to stop retransmitting, or (iii) another leader gets elected. A value has been chosen in the first case since correct learners only acknowledge after some value was chosen. In the second case, the leader election protocol ensures that a new leader is eventually elected. In the third case, the new leader will continue consensus and thus continue retransmission, unless it gets an acknowledgment from a correct leader as in the first case.

We now prove the lemma by contradiction. Suppose that no value is chosen. Eventually, no correct proposer suspects a correct leader (Lemma 10). Faulty leaders are eventually suspected by all correct proposers (Lemma 9), and a new leader is elected. Since leaders are elected in round-robin fashion and there is at least one correct proposer, eventually the correct proposer is elected and not suspected. The correct leader runs the recovery protocol and eventually gathers a change voucher. The leader proposes the value

returned by the change voucher, or its own value if the change voucher failed. In either case, the protocol ensures that correct acceptors will accept the value v accompanied with the change voucher. Since by assumption a vaulue is never chosen, the correct leader retransmits forever; since links are fair, all correct acceptors eventually receive and accept v. By definition, v is chosen. □

4.5 FaB Consensus Wrap-Up

We obtain the full FaB Paxos protocol by assembling all the sub-protocols described in the previous sections. Safety holds even in the weakest model, when as many as f proposers, f acceptors, and f learners are Byzantinely faulty. Even if the system starts in an asynchronous phase in which links are fair, asynchronous and authenticated, we assume that there exists some time T after which correct replicas and the links between them are timely. The model assumes that the stable phase lasts forever, although in practice it is sufficient that the stable phase lasts long enough for consensus to terminate.

The protocol makes progress as soon as everyone agrees on a leader that makes progress. At the latest, this happens at time T. Consensus terminates in two communication steps from then on.

5 FaB State Machine

In this section we show how to build a fast Byzantine replicated state machine using FaB Paxos. We follow the basic Paxos methodology: the acceptors determine the order in which a client's request is processed by the learners, which then respond to the client. A different instance of consensus is run for each "slot" in the sequence of requests (so the first instance determines which request is executed first, and so on). Each message is labeled with the instance of consensus that it belongs to.

There is one challenge, however: a faulty leader could poison a large number of instances of consensus, making recovery difficult. Furthermore, a new correct leader may not be able to determine which instances of consensus have been poisoned. We use a rate-limiting protocol to control the number of consensus protocols that are running in parallel. The new leader can then easily run recovery on all of these instances of consensus and then resume fast operation.

5.1 Rate limiting protocol

This protocol ensures that no more than α instances of consensus run in parallel.

After learners learn a value, they go through a *confirmation phase* by sending a confirmation message to a responsive quorum of $4f + 1$ acceptors and waiting for an acknowledgment in return. Acceptors do not send the acknowledgment right away: they wait until they receive a confirmation from $2f + 1$ learners. Then they consider the proposal *confirmed* and acknowledge the learners. The acceptors ignore all messages labeled last_confirmed$+1+\alpha$ or later. Because the response can be sent to the client before the confirmation phase, the rate limiting protocol does not add a communication step before the client gets a reply.

5.1.1 Correctness of rate limiting

The safety proofs for CS1-CS3 and the liveness proof CL2 are unmodified. We first show that the confirmation phase terminates.

Lemma 14. *When links between correct replicas are timely and the leader is correct, correct learners eventually receive $4f + 1$ acks to their confirmation message.*

Proof. If the leader is correct, then eventually all correct learners learn of the chosen value v. These correct learners send notification to all acceptors and wait for an acknowledgment from $4f + 1$ of them. Since there are at least $2f + 1$ correct learners who are sending confirmation messages to all acceptors, eventually correct acceptors will receive confirmation from $f + 1$ learners and will start sending acknowledgments to the learners. Since there are $4f+1$ correct acceptors, the learners will eventually receive $4f + 1$ acks and terminate the confirmation phase. □

Lemma 15 (CL1). *In the common case, a proposed value is eventually chosen.*

Proof. We prove the lemma by induction. The first instance of consensus is not affected by the rate limiting protocol, thus it will choose a value and confirm. If consensus instance x confirms,

then consensus instance $x + 1$ will accept messages and will eventually choose a value and confirm. Thus, in the common case all instances of consensus eventually choose a proposed value. □

5.2 Optimizations

Our FaB state machine can be optimized in several ways. Because of space limitation we can only sketch them here—they are presented in full in a technical report [15]. [*Note to the reviewers: we have included the optimizations in appendix*]

Tentative executions Castro and Liskov's PBFT [2] introduces *tentative executions* as a way to reduce the number of communication steps required by Byzantine state machine replication without reducing the number of communication steps used by PBFT's consensus protocol. The idea is to let replicas execute clients' requests before consensus is reached, tentatively trusting the information provided by the leader. If all replicas send to the client, together with their reply, the information that the protocol requires to complete consensus, the client can perform the last step of the consensus protocol at the same time as the replicas and verify whether trust in the leader was well put. In case of conflict, tentative executions are rolled back and the requests are eventually re-executed in the correct order. With this optimization, PBFT uses (in the common case) three communication steps from the moment the primary receives the request until the client receives a correct reply. The same optimization can applied to the FaB state machine, reducing to just two the number of communication steps required in the common case between the leader receiving the request and the client receiving the reply.

$2f + 1$ **Learners** One of the benefits of physically separating agreement from execution [19] is the opportunity to reduce the number of learners from $3f + 1$ to $2f + 1$, thereby reducing the required number of independently failing implementations of a given service.

To reduce the number of learners required in FaB Paxos, we modify the retransmission protocol. With only $2f+1$ learners, f of whom may be faulty, the retransmission protocol must be able to stop after the leader receives a response from $f + 1$ learners, only one of which may be correct.

Out-of-date learners running the pull protocol must be able to identify correct learners. We use signatures to provide evidence before retransmission stops: the learners sign their results and exchange signatures. The resulting group of signatures can be used by a learner to vouch for its reply's correctness. Unless we are careful, gathering these signatures could add an additional communication step to the protocol. Fortunately, the reply can be sent to the client before generating and distributing the signatures: the client will be able to identify correct responses, since these will be vouched by at least $f + 1$ learners.

6 Conclusions

FaB Paxos is the first Byzantine Paxos protocol to achieve consensus in just two communication steps in the common case, matching the lower bound on the number of communication steps for the crash failure model. This advantage comes at the cost of requiring a significantly higher number of acceptors than those needed by slower Byzantine Paxos protocols. These extra acceptors are precisely what allows a newly elected leader in FaB Paxos to determine, via the change vouchers, whether or not a value had already been chosen for the current instance of consensus under the supervision of a previous leader—a key property to guarantee the safety of FaB Paxos in the presence of failures.

In traditional state machine architectures, the costs of this additional replication would make FaB Paxos unattractive for all but the applications most committed to reducing latency. In the new state machine architecture that we have recently proposed, however, acceptors are significantly cheaper to implement [19], making the design point occupied by FaB Paxos much more intriguing.

References

[1] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Reconstructing paxos. *SIGACT News*, 34(2), 2003.

[2] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, 1999.

[3] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proc. 4th OSDI*, pages 273–287, 2000.

[4] P. Dutta, R. Guerraoui, and M. Vukolić. Asynchronous byzantine consensus: Complexity, re-

silience and authentication. Personal Communication, September 2004.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

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

[7] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults. Technical Report MIT-LCS-TR-821, 2001.

[8] K. Kursawe. Optimistic byzantine agreement. In *Proc. 21st SRDS*, 2002.

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

[10] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[11] L. Lamport. Lower bounds for asynchronous consensus. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, June 2002.

[12] L. Lamport. Lower bounds for asynchronous consensus. In *LNCS 2584*, pages 22–23. Springer, 2003.

[13] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[14] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing 11/4*, pages 203–213, 1998.

[15] J-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. Technical Report TR04-08, The University of Texas at Austin, 2004.

[16] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Distributed Computing, 16th international Conference, DISC 2002*, pages 311–325, October 2002.

[17] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. 18th SOSP*, Oct. 2001.

[18] F. B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[19] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267. ACM Press, 2003.

A Consensus

A.1 Failure Detector

The failure detector has the following two properties: (1) a leader that makes no progress is eventually suspected by all correct proposers, and (2) when links are timely, eventually no correct proposer suspects a correct leader. Our failure detector is based on a classical time-out mechanism.

After learners learn a value, they send an acknowledgment message to all proposers. We use that message to detect progress.

Each proposer p starts a timer at the beginning of the consensus protocol (either when consensus is started explicitely, or triggered by a retransmission from a client who is waiting for a response). If it has received fewer than $f + 1$ acknowledgments when the timer expires, then p suspects the current leader and doubles the time-out delay. The time-out delay is eventually large enough that no correct leader is suspected.

Lemma 16. *A leader that makes no progress is eventually suspected by all correct proposers.*

Proof. If a leader makes no progress, then no correct learner will send an acknowledgment, and therefore proposers will receive at most f acknowledgments (from the malicious learners). Since all proposers receive fewer than $f + 1$ acknowledgments, all correct proposers will suspect the leader. □

Lemma 17. *Eventually, no correct proposer suspects a correct leader.*

Proof. The system eventually reaches the stable phase where links are timely, and time-outs grow until eventually they are large enough that no message is incorrectly considered lost. Then, correct leaders are never suspected because all proposers receive the required $f+1$ acknowledgments in time. Since the leader is correct and links are timely, all correct learners learn the value proposed by the leader and send an acknowledgment to all proposers. These acknowledgments are not lost and arrive in time because links are timely and time-outs have grown enough. Thus, each correct proposer will receive at least $2f + 1$ acknowledgments. □

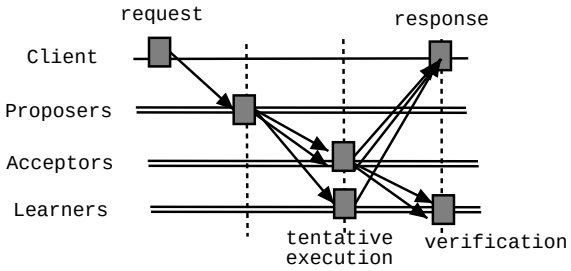

Figure 2: Optimized FaB state machine

B State Machine Optimizations

B.1 Tentative Execution

Figure 2 shows the operation of a FaB state machine that uses tentative execution. In a straightforward implementation of state machine replication, learners do not process a client's request or reply to the client until they receive from the acceptors the relative order in which that request should be executed. In contrast, with tentative execution the leader sends the request and a suggested execution order to the learners directly. The learners tentatively execute the request in the suggested order, and reply to the client, without waiting to hear from the acceptors whether the execution order suggested by the leader is correct.

Eventually, the learners do receive from the acceptors the information necessary to either confirm or repudiate the tentative execution order. In the latter case, the learners roll back the affected requests. Acceptors send the same information also to the client, who can then determine whether or not the reply received as a result of tentative execution will be rolled back.

B.2 $2f + 1$ Learners

We show how to reduce the number of learners to $2f + 1$ without delaying the replies to clients. This optimization requires some communication and the use of signatures in the common case, but still manages to answer queries within four communication steps (or three with tentative execution).

In the $2f + 1$ version of FaB Paxos, learners must satisfy two requirements: (i) $f + 1$ correct learners must communicate with the client so that the client can identify the correct reply, and (ii)

eventually all correct learners must agree on the ordering. These requirements are discharged by the push and pull protocols, respectively. We need to modify these protocols in order for them to continue to function despite the reduction in the number of learners.

In the original push protocol, the leader retransmits the query until it gets an acknowledgment from $2f + 1$ learners, ensuring that $f + 1$ of them are correct. When there are only $2f + 1$ learners in total, f of whom possibly faulty, the leader may never receive this many acknowledgments. We therefore change the push protocol so that it ends as soon as the leader gets $f + 1$ acknowledgments—retransmission may now stop when only a single correct learner knows the correct response.

In order for the pull protocol to be able to pick up where the push protocol left, that single correct learner must be able to convince other learners that its reply is correct. We therefore strengthen the push protocol's postcondition by adding information in the acknowledgments. In addition to the client's request and reply obtained by executing that request, acknowledgments now also contain $f+1$ signatures from distinct learners that verify the same reply.

After computing the reply to the client's request, learners now sign it and send that signature to all learners, expecting to eventually receiving $f+1$ signatures that verify their reply. Since there are $f + 1$ correct learners, each is guaranteed to be able to eventually gather a complete acknowledgment that will satisfy the leader. The leader is then assured that at least one of the learners who sent it a valid acknowledgment is correct and will support the pull protocol.

In the pull protocol, learners periodically query for each other's acknowledgments. They learn value v if they receive from any of their peers an acknowledgment for v with $f + 1$ valid signatures. Thus, eventually all correct learners learn the value v.

Learners now send their acknowledgment signatures to every learner and wait to receive $f + 1$ matching valid signatures. This could slow down the response to the clients, but we observe that it is safe for learners to send their response directly to the client before computing the signature or sending it to their peers. Clients can already distinguish correct replies from incorrect ones since only correct replies are vouched for by $f+1$ learners.

B.3 Rejoin

One possible improvement to our protocol is to allow repaired servers to rejoin the system (for example a crashed node that was rebooted). This allows our system to tolerate more faults, as long as at any point in time no more than f servers are either faulty or rejoining.

The rejoin protocol must restore the replicas's state, and as such it is different depending on the role that the replica plays. Proposers do not have much state, although it is useful to know who the leader is. Therefore, a joining proposer queries a quorum of acceptors for their current proof-of-leadership and adopts the largest valid response.

Acceptors must never accept two different values for the same proposal number. In order to ensure that this invariant holds, a rejoining acceptor queries the other acceptors for the last confirmed decree d, and it then ignores all decrees until $d+\alpha$. Once the system moves on to this instance of consensus, the acceptor has completed its rejoin.

The state of the learners consists of the list of executed operations (more precisely, the state resulting from executing all these operations). A rejoining learner therefore queries a quorum of other learners for the list of executed operations. Checkpoints could be used for faster state transfer as has been done before [2, 10].

C Approximate Theorem Counterexample

Lamport's "approximate theorem" 3a [12] gives a lower bound on the number of acceptors for two-step consensus. Our phrasing of this theorem differs from his because there is a counterexample to the approximate theorem.

The "approximate theorem" 3a reads: "If there are at least two proposers whose proposals can be learned with a 2-message delay despite the failure of Q acceptors, or there is one such possibly malicious proposer that is not an acceptor, then $N > 2Q+F+2M$." N is the number of acceptors.

M is the number of failures despite which safety must be ensured. F is the number of failures despite which liveness must be ensured. We believe that the statement is correct, except in the corner case where no learner can fail. In that case, it is possible to use only $3f + 1$ acceptors to tolerate f Byzantine failures and be able to learn in two message delays despite up to f Byzantine failures (3a claims that $5f+1$ acceptors are needed in that case). Learners learn v if $2f+1$ acceptors say they have accepted it. Since any two quorums of $2f+1$ intersect in a correct acceptor, no two learners will learn different values. If the leader is faulty then it is possible that no value gets learned. In that case we go through a leader election, and the new leader asks the learners for the value that they have learned. Since learners are all correct, the new leader can wait for all learners to reply with a signed response. The leader can therefore choose a value to propose that will maintain the safety properties. Since the learners' answers are signed, the new leader can forward them to the acceptors to convince them to accept a new value.

D Pseudocode

Common Case Algorithm

```
client.invoke(operation) {
        ts++
        r := Q-RPC(1,Proposers,("invoke",operation,ts
             ))
        return r
}

proposer.onInvoke(sender, operation, ts) {
        if client-ts[sender]>=ts then
                reply with last-reply[sender]
                return
        client-ts[sender] := ts // only one
             concurrent call per client
        if (I-am-leader()) then
           decree++
           firstOf( { propose(sender, operation) }
                  , { wait until (not I-am-Leader())
                        }
           )
        else
                send ("invoke-forward",sender,
                     operation,ts) to current-leader
                     ()
                // retransmission handled by the
                     client
}

proposer.propose(sender,operation) {
        R := open-jaw-Q-RPC(4f+1, Acceptors, 2f+1,
             Learners, ("propose",decree,0,operation,
             pol) )
        // returns after receiving an ACK from 2f+1
             different learners
        reply := value present f+1 times in R
        last-reply[sender] := reply
        send reply to sender
}

proposer.onInvokeForward(sender,client,operation,ts)
     {
        onInvoke(client,operation,ts);
}

proposer.onDenied(sender,proposal,newPol) {
        if pol<newPol then pol := newPol
}

// psn stands for proposal sequence number
acceptor.onPropose(sender,decree,psn,value,pol,[
     optional] change-voucher) {
        if elects(pol)!=sender then drop request
        max-pol := max( max-pol, pol )
        if last-accepted-proposal[decree]>psn or pol<
             max-pol then
             send ("denied",last-accepted-proposal[
                  decree],max-pol) to sender
             return
        if proposal[decree]!=\bot and proposal[decree
             ]!=value
        and not allow-change(decree,psn,value,change-
             voucher) then
             send ("denied",last-accepted-proposal[
                  decree],max-pol) to sender
             return
        proposal[decree]:=value
        last-accepted-proposal[decree] = psn
        send ("accepted",decree,psn,value,elects(max-
             pol)) to all learners
        reply with "propose-ok" to sender
}

acceptor.allow-change(decree,psn,value,change-voucher
     ) {
        // change is introduced in later versions of
             the protocol
        return false
}

learner.onAccepted(sender,decree,psn,value,active-
     leader) {
        if decision[decree] != \bot then
            poke; drop request  // postcondition
                 already holds; nothing to do
        val[sender][decree][psn] := value
        regent[decree] := max( regent[decree], active
             -leader )
        if \exists decree d and \exists psn p such
             that 4f+1 of the val[][d][p] have the
             same value v then
           decision[decree] := v
           responder[decree].poke()
        // no direct reply, but the responder will
             when we've made a decision
}

// learners maintain an array of active objects
     called responders: one per decree.

learner.responder[decree].run() {
        responder[decree].executeWhenReady()
}

learner.responder[decree].executeWhenReady() {
        wait until decision[decree] != \bot
        wait until last-executed == decree - 1,
             periodically calling waitingGap
        result[decree] := execute( decision[decree] )
        last-executed := decree
        poke()
}

// called when a value was chosen for some decree d
// but not value was chosen for d', d'<d
learner.waitingGap() {
  // do nothing
}

learner.responder[decree].poke() {
        if result[decree] == \bot then return
        send ("result",result) to client once
        send ("ack-from-learner",decree,result) to
             regent[decree] once
}
```

Pull Protocol

```
// query neighbors when there is a gap in the
     decision sequence
learner.waitingGap() {
  next := last-executed + 1
  if (decision[next] != \bot) return;
  send ("pulling-decision",next) to all learners once
}
```

```
learner.onPullingDecree(sender,decree) {
  if (decision[decree]!=\bot) reply with ("decision",
      decree,decision[decree]);
}

// decide v if f+1 other learners have decided v
learner.onDecision(sender,decree,value) {
  learner-says[decree][value] union= sender
  if there is a decree d and a value v s.t. |learner-
      says[d][v]|>f and decision[d]==\bot
  then {
       decision[d] := v;
       responder[d].poke();
  }
}
```

Rate Limiting Algorithm

```
// confirmed_i[decree] <=>
//     acceptor i got an Ack for decree 'decree' from
     2f+1 learners
//     OR acceptor i knows that some correct
    acceptors j has confirmed_j[decree]
//     OR decree is negative
//
// reported_j[decree] =>
//     learner j has decision[decree] != \bot and
    that decision was communicated
//     to 4f+1 acceptors who responded (with ack-
    confirmed)
//     OR decree is negative

acceptor.onPropose(sender,decree,psn,value,pol) {
        for each j \in [decree - alpha,decree - 1] s.
             t. not confirmed[j]:
                Q := Q-RPC(3f+1,Acceptors, ("get-
                     confirm",j) )
                if Q contains f+1 "yes" then
                     confirmed[j] := true
        if exists j \in [decree - alpha,decree - 1] s
            .t. not confirmed[j] then
                drop request;
        super()
}

acceptor.onGetConfirm(sender, decree) {
        reply with confirmed[decree]
}

acceptor.onAckFromLearner(sender,decree) {
        Acks[decree] := Acks[decree] union sender
        if |Acks[decree]| >= 2f+1 then
                confirmed[decree]:=true
                reply with "ack-confirmed"
        drop request
}

learner.onAccepted(sender,decree,psn,value,active-
    leader) {
        if one of reported[decree-1] ... reported[
            decree - alpha] is false then
                drop request;
        super()
}

learner.responder[decree].run {
        parallel( { executeWhenReady() }
                , { // report on execution
                    wait until decision[decree] != \
                         bot
                    Q-RPC(4f+1,Acceptors, ("ack-from-
                         learner",decree))
                    reported[decree] := true }
        )
}
```

Recovery Algorithm

```
// puts the funky register back in a good state
proposer.recover() {
        assert( elects(pol) == self )
        Q := Q-RPC(3f+1, Acceptors, ("get-latest-
            confirmed-decree",pol))
        lastest-confirmed := f+1th largest value in Q
        for decree := latest-confirmed to latest-
            confirmed+3 (excluded)
              complete-decree(decree)
        decree := latest-confirmed + 2
}

// ensures that a decision has been reached on that
    particular decree
proposer.complete-decree(decree) {
        ack[] := array of 3f+1 values, all "false"
        triplet[] := arrray of 5f+1 quadruplets: (
            decree,psn,value,sig) (initally all \bot
            )
        max-p := 0
        repeat
                send ("read-signed-decree-and-advance
                    ",decree,max-p,pol) to all
                    acceptors i such that triplet[i
                    ]==\bot or triplet[i].psn<max-p
                for each ("read-signed-decree-and-
                    advance-ok",decree,triple) in
                    the input queue from sender
                                if triple.sig is valid
                                    and triple.psn<=
                                    max-p then triplet
                                    [sender]:=triple
                max-p := largest psn in triplet[]
                Proof := { triplet[i] | triplet[i].
                    psn==max-p }
                if |Proof|>=4f+1 then
                                val := value in 2f+1
                                    elements of Proof
                                    , or "no-op" if
                                    there's none
                                send ("propose",decree,
                                    max-p+1,val,pol,
                                    proof) to all
                                    Acceptors
                                for each ("ack-from-
                                    learner",decree,r)
                                     from sender
                                         ack[sender
                                             ] := true
        until |ack[]|>=2f+1
}

// signed read, plus guarantee that we'll ignore
    earlier regents
acceptor.onReadSignedDecreeAndAdvance(sender, decree
    , psn, pol) {
        if (elects(pol)!=sender) then drop request
        max-pol := max( max-pol, pol )
        if max-pol==pol and last-accepted-proposal[
            decree]<psn then
                last-accepted-proposal[decree]:=psn
        rep := (proposal[decree],decree,last-accepted
            -proposal[decree])
        reply with (rep, signature(rep))
}

acceptor.onGetLatestConfirmedDecree(sender,pol) {
        if (elects(pol)!=sender) then drop request
        reply with largest x such that confirmed[x]
            is true
}

acceptor.allow-change(decree,psn,value,change-voucher
    ) {
        if change-voucher contains 4f+1 validly
            signed (decree_i,psn_i,value_i) triplets
             such that
           decree_i==decree and psn_i==psn - 1 and
                last-accepted-proposal<=psn_i for all
                 i and
           (either 2f+1 of the entries have value_i==
                value or no 2f+1 entries have the
                same value)
        then return true
        else return false
}
```

Leader Election Algorithm

```
regency(pol) {
        return the smallest of the 2f+1 signed
            numbers in pol
}

elects(pol) {
        return regency(pol) mod (the number of
            proposers)
}

// valid pol contains 2f+1 signed votes that elect
    the same leader
valid(pol) {
```

```
        return true iff pol contains 2f+1 numbers n_i
             each signed by a different proposer
        and there exists x s.t. forall n_i in pol : x
             == n_i mod (the number of proposers)
}

// called when no progress is observed
proposer.noProgress() {
        let suspected-leader := elects(pol)
        let advance := 1
        firstOf(
                { repeat forever {
                                vote := sign( regency
                                     (pol) + advance
                                     )
                                send ("vote",vote) to
                                     vote mod (
                                     number of
                                     proposers)
                                wait for timeout
                                     seconds
                                advance := advance
                                     + 1
                                timeout := timeout
                                     * 2
                } },
                { repeat forever {
                                process incoming "new
                                     -leader"
                                     messages
                } },
                { wait until suspected-leader <
                     elects(pol) }
        )
        // at this point, a new leader has been
             elected
}

proposer.onNewLeader(sender, newPol) {
        if regency(newPol)>regency(pol) then
                pol := newPol
        // no reply necessary
}

proposer.onVote(sender, vote) {
        votes[sender] := vote
        let votesForMe := largest 2f+1 elements in
             votes[]
        if (pol < votesForMe) then
                        pol := votesForMe // I become
                             the leader
        if (vote<regency(pol)) or (IamLeader()) then
                        send ("new-leader", pol) to
                             sender
}

proposer.IamLeader() {
        if elects(pol)==me then return true
        else return false
}
```

2f+1 learners (optional)

```
// only report after we have f+1 valid signatures
learner.responder[decree].run {
        parallel( { executeWhenReady() }
                , { // report on execution
                    wait until decision[decree] != \
                         bot
                    and there are f+1 values j s.t.
                         signature[decree][j] != \bot
                    Q-RPC(4f+1,Acceptors, ("ack-from-
                         learner",decree))
                    reported[decree] := true }
        )
}

// sign, and send signature to fellow learners too
learner.responder[decree].poke() {
        if result[decree] == \bot then return
        send (sign("ack-from-learner",decree,result))
             to all learners once
        if there are f+1 values j s.t. signature[
             decree][j] != \bot then
                send ("ack-from-learner",decree,
                     result) to regent[decree] once
}

// store signatures
learner.onAckFromLearner(sender,decree,result,
     signature) {
        signature[decree][sender] := ("ack-from-
             learner",decree,result,signature)
}

// leader can now stop retransmission ("push") once
     it has f+1 identical replies
proposer.propose(sender,operation) {
        R := open-jaw-identical-Q-RPC(4f+1, Acceptors
             , f+1, Learners, ("propose",decree,0,
             operation,pol) )
        // returns after receiving an identical reply
             from f+1 different learners
        reply := value present f+1 times in R
        last-reply[sender] := reply
        send reply to sender
}

//
// With 2f+1 learners, the pull protocol must change
     as follows
//

// query neighbors when there is a gap in the
     decision sequence
learner.waitingGap() {
  next := last-executed + 1
  if (decision[next] != \bot) return;
  send ("pulling-decision",next) to all learners once
}

learner.onPullingDecree(sender,decree) {
  if (decision[decree]!=\bot)
  and there are f+1 values j s.t. signature[decree][j
       ] != \bot then
        reply with ("decision",decree,decision[decree
             ],signature[decree][]);
}

// decide v if a learner has f+1 signatures for v
learner.onDecision(sender,decree,value,sigs[]) {
  if there are f+1 values j s.t. sigs[j] != \bot then
       {
       decision[decree] := value;
       responder[decree].poke();
  }
}
```

Helper Functions

```
parallel(code block, code block, ...)
  executes all code blocks in parallel and wait until
       all return.

firstOf(code block, code block, ...)
  executes all code blocks in parallel.
  As soon as one of the blocks returns, kill all the
       other blocks and return

There is an ordering relation on the proof-of-
     leaderships

elects(proof-of-leadership)
  returns the name of the proposer who is vouched for
       by this proof-of-leadership

signature(anything)
  returns a valid digital signature for the arguments
       (using the caller's key pair)
```