Fast Byzantine Consensus

Jean-Philippe Martin and Lorenzo Alvisi, *Senior Member*, *IEEE*

Abstract—We present the first protocol that reaches asynchronous Byzantine consensus in two communication steps in the common case. We prove that our protocol is optimal in terms of both number of communication steps and number of processes for two-step consensus. The protocol can be used to build a replicated state machine that requires only three communication steps per request in the common case. Further, we show a parameterized version of the protocol that is safe despite f Byzantine failures and, in the common case, guarantees two-step execution despite some number t of failures ($t \leq f$). We show that this parameterized two-step consensus protocol is also optimal in terms of both number of communication steps and number of processes.

Index Terms—Distributed systems, Byzantine fault tolerance, consensus.

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 [12]. A single process can be a member of more than one class. Consensus can be specified using the following three safety properties and two liveness properties:

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

CS2. 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 [12], 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 [11], [22], 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 [5]. 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 as a leader, with the responsibility of communicating with the acceptors. Paxos guarantees progress only when the leader is unique and can communicate in a timely manner with sufficiently many acceptors, but it ensures safety even with no leader or with multiple leaders. Our protocol follows a similar structure, but we choose the leader from the proposers instead of the acceptors.

Paxos is also attractive because it can be made very efficient in *gracious executions*, i.e., executions where 1) there is a unique correct leader, 2) all correct acceptors agree on its identity, and 3) the system is in a period of synchrony. Note that processes other than the leader may still fail during gracious executions. 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, the FastPaxos [1] protocol by Boichat et al. only requires two communication steps[1] in a gracious execution to reach consensus in non-Byzantine environments, matching the lower bound formalized by Keidar and Rajsbaum [8] (FastPaxos should not be confused with Lamport's more recent "Fast Paxos" (with a space) [15] that uses a different approach to reduce the number of communication steps in the common case). 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 Paxos to support Byzantine fault-tolerant state machine replication. The resulting systems perform surprisingly well: They add modest latency [2], can proactively recover from faults [3], can make use of existing software diversity to exploit opportunistic N-version programming [20], and can be engineered to protect confidentiality and reduce the replication costs incurred to tolerate f faulty state machine replicas [23].

These 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 Paxos needs a

• *The authors are with the Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX 78712-0233. E-mail: {jpmartin, lorenzo}@cs.utexas.edu.*

Manuscript received 16 Sept. 2005; revised 21 Dec. 2005; accepted 21 Mar. 2006; published online 3 Aug. 2006.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSCSI-0128-0905.

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

1545-5971/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

minimum of three additional communication steps (rather than the two required in the non-Byzantine case) before the request can be executed.[2]

We make three contributions. First, we introduce Fast Byzantine (or FaB) Paxos, a Byzantine consensus protocol that completes in two communication steps in the common case (we say it is *two-step*), without using expensive digital signatures. FaB Paxos requires $5f + 1$ acceptors and tolerates f Byzantine faults. Second, we show a generalization of FaB Paxos—Parameterized FaB Paxos—that requires $3f + 2t + 1$ acceptors to tolerate f Byzantine failures and is two-step as long as at most t acceptors fail. Third, we show that both FaB Paxos and Parameterized FaB Paxos are tight in the sense that they use the minimal number of processes required for two-step protocols.

Since building a replicated state machine from consensus adds a single communication step, FaB Paxos or Parameterized FaB Paxos can be used to build a Byzantine fault-tolerant replicated state machine that requires only three communication steps per operation in the common case. By comparison, Castro and Liskov's Practical Byzantine Fault-tolerance protocol [3] uses four communication steps in the common case.[3]

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 FaB Paxos 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 [20]. However, an architecture for Byzantine fault tolerant state machine replication that physically separates agreement from execution [23] makes this trade-off look much more attractive. In this 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.

Decoupling agreement from execution leads to agreement replicas (i.e., 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, such acceptor 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: We discuss related work in Section 2 and our system model in Section 3. We present *f*-tolerant FaB Paxos in Section 4 and generalize it to Parameterized FaB Paxos in Section 5. We give lower bounds on the number of processes in Section 6, showing that both protocols are optimal. We show in Section 7 how to build a replicated state machine from our consensus protocols, then present some optimizations in Section 8 before concluding.

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 [12] and Castro and Liskov's work on Practical Byzantine Fault-tolerance (PBFT protocol) [3]. 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 [17].

The two earlier protocols that are closest to FaB Paxos are the FastPaxos protocol by Boichat and colleagues [1], and Kursawe's Optimistic asynchronous Byzantine agreement [10]. Both protocols share our basic goal: to optimize the performance of the consensus protocol when runs are, informally speaking, well-behaved.

The most significant difference between FastPaxos and FaB Paxos 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 $5f + 1$ necessary for FaB Paxos. A subtler difference between the two protocols pertains to the conditions under which FastPaxos achieves consensus in two communication steps: 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 weaker than these, in that during gracious executions, processes *can* fail, provided that the leader does not fail. As a final difference, FastPaxos does not rely, as we do, on eventual synchrony but on an eventual leader oracle; however, since we only use eventual synchrony for leader election, this difference is superficial.

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 well-behaved execution 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.

There are also protocols that use failure detectors to complete in two communication steps in some cases. Both the SC protocol [21] and the later FC protocol [7] achieve this goal when the failure detectors make no mistake and the coordinator process does not crash (their coordinator is similar to our leader). FaB Paxos differs from these protocols because it can tolerate unreliable links and Byzantine failures. Other protocols offer guarantees only

2. No protocol can guarantee to take fewer than two rounds to reach Byzantine consensus. This bound holds even in a synchronous system, where one process may crash [16].

3. Even with the tentative execution optimization (see Section 7).

for certain initial configurations. The oracle-based protocol by Friedman et al. [6], for example, can complete in a single communication step, if all correct nodes start with the same proposal (or in a variant that uses $6f + 1$ processes, if at least $n - f$ of them start with the same value and are not suspected). FaB Paxos differs from these protocols in that it guarantees learning in two steps regardless of the initial configuration.

In a paper on lower bounds for asynchronous consensus [13], Lamport conjectures in "approximate theorem" 3a the existence of a bound $N > 2Q + F + 2M$ on the minimum number N of acceptors required by two-step Byzantine consensus, where 1) F is the maximum number of acceptor failures despite which consensus liveness is ensured; 2) M is the maximum number of acceptor failures despite which consensus safety is ensured, and 3) Q is the maximum number of acceptor failures despite which consensus must be two-step. Lamport's conjecture is more general than ours—we do not distinguish between M, F, and Q—and more restrictive—unlike us, Lamport does not consider Byzantine learners but instead assumes that they can only crash. This can be limiting when using consensus for the replicated state machine approach: The learner nodes execute the requests, so their code is comparatively more complicated and more likely to contains bugs that result in unexpected behavior. Lamport's conjecture does not technically hold in the corner case where no learner can fail.[4] Dutta et. al have recently derived a comprehensive proof of Lamport's original conjecture under the implicit assumption that at least one learner may fail [4]. In a later paper [14], Lamport gives a formal proof of a similar theorem for crash failures only. He shows that a protocol that reaches two-step consensus despite t crash failures and tolerates f crash failures requires at least $f + 2t + 1$ acceptors. In Section 5, we show that in the Byzantine case, the minimal number of processes is $3f + 2t + 1$.

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 [12], we describe the behavior of FaB Paxos in terms of the actions performed by proposers, acceptors, and learners. We assume that the number n of processes in the system is large enough to accommodate $3f + 1$ proposers, $5f + 1$ acceptors, and $3f + 1$ learners. Note that a single process may play multiple roles in the protocol. Up to f of the processes playing each role may be Byzantine faulty. 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. Unlike [13], we allow learners to fail in a Byzantine manner.

4. The counterexample can be found in our technical report [18].

variable	initial	comment
Globals		
p, a, l		Number of proposers, acceptors, learners
f		Number of Byzantine failures tolerated
Proposer variables		
Satisfied	$\emptyset$	Set of proposers that claim to be satisfied
Learned	$\emptyset$	Set of learners that claim to have learned
Acceptor variables		
accepted	$(\perp, \perp)$	Value accepted and the corresponding proposal number
Learner variables		
learner.accepted[j]	$(\perp, \perp)$	Value and matching proposal number acceptor j says it accepted
learner.learn[j]	$(\perp, \perp)$	Value and matching proposal number learner j says it learned
learner.learned	$(\perp, \perp)$	Value learned and the corresponding proposal number

Fig. 1. Variables for the FaB pseudocode.

FaB Paxos does not use digital signatures in the common case; however, it does rely on digital signatures when electing a new leader. All acceptors have a public/private key pair—we assume that all proposers and acceptors know all public keys and that correct acceptors do not divulge their private key. We also assume that Byzantine processes are not able to subvert the cryptographic primitives.

Since it is impossible to provide both safety and liveness for consensus in the asynchronous model [5], we ensure safety at all times and only guarantee liveness during periods of synchrony.

4 Fast Byzantine Consensus

We now present FaB Paxos, a two-step Byzantine fault-tolerant consensus protocol that requires $5f + 1$ processes—in Section 6, we show that this number is optimal. More precisely, FaB Paxos requires $a \geq 5f + 1$ acceptors, $p \geq 3f + 1$ proposers, and $l \geq 3f + 1$ learners; as in Paxos, each process in FaB Paxos can play one or more of these three roles. We describe 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.

4.1 The Common Case

We first describe how FaB Paxos works in the common case, when there is a unique correct leader, all correct acceptors agree on its identity, and the system is in a period of synchrony.

FaB is very simple in the common case, as can be expected by a protocol that terminates in two steps. Fig. 1 shows the variables we use, and Fig. 2 shows the protocol's pseudocode. The `pnumber` variable (proposal number) indicates which process is the leader; in the common case, its value will not change. The code starts executing in the `onStart` methods. In the first step, the leader proposes its value to all acceptors (line 3). In the second step, the acceptors accept this value (line 21) and forward it to the learners (line 22). Learners learn a value v when they observe that $\lceil (a + 3f + 1)/2 \rceil$ acceptors have accepted the value (line 25). In the common case, the time-out at line 12

```
1  leader.onStart():
2    // proposing (PC is null unless recovering)
3    send (PROPOSE, value, pnumber, PC) to all acceptors
4    until |Satisfied| >= ⌈(p + f + 1)/2⌉
5
6  proposer.onLearned(): from learner l
7    Learned := Learned union {l}
8    if |Learned| >= ⌈(l + f + 1)/2⌉ then
9      send (SATISFIED) to all proposers
10
11 proposer.onStart():
12   wait for timeout
13   if |learned| < ⌈(l + f + 1)/2⌉ then
14     leader-election.suspect( leader-election.getRegency() )
15
16 proposer.onSatisfied(): from proposer x
17   Satisfied := Satisfied ∪ {x}
18
19 acceptor.onPropose(value, pnumber, progcert): from leader
20   if not already accepted then
21     accepted := (value, pnumber) // accepting
22   send (ACCEPTED, accepted) to all learners
23 learner.onAccepted(value, pnumber): from acceptor ac
24   accepted[ac] := (value, pnumber)
25   if there are ⌈(a + 3f + 1)/2⌉ acceptors x
26   such that accepted[x] == (value, pnumber) then
27     learned := (value, pnumber) // learning
28     send (LEARNED) to all proposers
29
30 learner.onStart():
31   wait for timeout
32   while (not learned) send (PULL) to all learners
33
34 learner.onPull(): from learner ln
35   If this process learned some pair (value, pnumber) then
36     send (LEARNED, value, pnumber) to ln
37
38 learner.onLearned(value, pnumber): from learner ln
39   Learn[ln] := (value, pnumber)
40   if there are f + 1 learners x
41   such that learn[x] == (value, pnumber) then
42     learned := (value, pnumber) // learning
```

Fig. 2. FaB pseudocode (excluding recovery).

will never trigger. We will use that code later; the leader election interface is given in Fig. 3. FaB avoids digital signatures in the common case because they are computationally expensive. Adding signatures would reduce neither the number of communication steps nor the number of servers since FaB is already optimal in these two measures.

We defer the full correctness proof for FaB until we have discussed the recovery protocol in Section 4.4—in the following, we give an intuition of the correctness argument.

Let correct acceptors only accept the first value they receive from the leader and let a value v be *chosen* if $\lceil (a + f + 1)/2 \rceil$ correct acceptors have accepted it. These two requirements are sufficient to ensure CS1 and CS2: Clearly, only a proposed value may be chosen and there can be at most one chosen value since at most one value can be accepted by a majority of correct acceptors. The last safety clause (CS3) requires correct learners to learn only a chosen value. Since learners wait for $\lceil (a + 3f + 1)/2 \rceil$ identical reports and at most f of those come from faulty acceptors, it follows that the value was necessarily chosen.

```
1  int leader-election.getRegency()
2      // return the number of the current regent (leader is regent % p)
3      // if no correct node suspects it then the regency continues.
4
5  int leader-election.getLeader():
6      return getRegency() % p
7
8  void leader-election.suspect(int regency)
9        // indicates suspicion of the leader for "regency".
10       // if a quorum of correct nodes suspect the same regency r,
11       // then a new regency will start
12
13 void leader-election.consider(proof)
14       // consider outside evidence that a new leader was elected
```

Fig. 3. Interface for leader election protocol.

4.2 Fair Links and Retransmissions

So far, we have assumed synchrony. While this is a reasonable assumption in the common case, our protocol must also be able to handle periods of asynchrony. We weaken our network model to consider fair asynchronous authenticated links (see Section 3). Note that now consensus may take more than two communication steps to terminate, e.g., when all messages sent by the leader in the first round are dropped.

Our end-to-end retransmission policy is based on the following pattern: The caller sends its request repeatedly, and the callee sends a single response every time it receives a request. When the caller is satisfied by the reply, it stops retransmitting. We alter the pattern slightly in order to accommodate the leader election protocol: Other processes must be able to determine whether the leader is making progress and, therefore, the leader must make sure that they, too, receive the reply. To that end, learners report not only to the leader but also to the other proposers (Fig. 2, line 28). When proposers receive enough acknowledgments, they are "satisfied" and notify the leader (line 9). The leader only stops resending when it receives $\lceil (p + f + 1)/2 \rceil$ such notifications (line 4). If proposers do not hear from $\lceil (l + f + 1)/2 \rceil$ learners after some time-out, they start suspecting the leader (line 14). If $\lceil (p + f + 1)/2 \rceil$ proposers suspect the leader, then a new leader is elected.[5] The retransmission policy, therefore, ensures that, in periods of synchrony, the leader will retransmit until it is guaranteed that no leader election will be triggered. Note that the proposers do not wait until they hear from all learners before becoming satisfied (since some learners may have crashed). It is possible therefore that the leader stops retransmitting before all learners have learned the value. To ensure that eventually

5. We do not show the election protocol, because existing leader election protocols can be used here without modification, e.g., the leader election protocol in [3].

all correct learners do learn the value, lines 30 to 42 of the protocol require all correct learners still in the dark to pull the value from their peers.

4.3 Recovery Protocol

Recovery occurs when the leader election protocol elects a new leader. Although we can reuse existing leader election protocols as-is, it is useful to go through the properties of leader election. The output of leader election is a regency number r. This number never decreases, and we say that proposer $r \bmod p$ is the leader. Each node in the system has an instance of a leader-election object, and different instances may initially indicate different regents. Nodes indicate which other nodes they suspect of being faulty; that is, the input to the leader election protocol. If no more than f nodes are Byzantine and at least $2f+1$ nodes participate in leader election, then leader election guarantees that if no correct node suspects the current regent, then eventually 1) all leader-election objects will return the same regency number and 2) that number will not change. Leader election also guarantees that if a quorum of correct nodes ($\lceil (p+f+1)/2 \rceil$ nodes out of p) suspects regent r, then the regency number at all correct nodes will eventually be different from r. Finally, leader election also generates a $proof_r$ when it elects some regent r. If $proof_r$ from a correct node is given to a leader-election object o, then o will elect regency $r', r \leq r'$.

The interface to leader election is shown in Fig. 3. `getRegency()` returns the current regency number, and `getLeader()` converts it to a proposer number. Nodes indicate their suspicion by calling `suspect(`r`)`. When leader-election elects a new leader, it notifies the node through the `onElected(`$regency, proof_r$`)` callback (not shown). If necessary, $proof_r$ can then be given to other leader-election objects through the `consider(`$proof_r$`)` method.

When proposers suspect the current leader of being faulty, they trigger an election for 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 CS2. Second, a previous malicious leader may have performed a *poisonous write* [19], i.e., a write that prevents learners from reading any value—for example, a malicious leader could propose a different value to each acceptor. If the new leader is correct, consensus in a synchronous execution should nonetheless terminate. In our discussion so far, we have required acceptors to only accept the first value they receive. If we maintained this requirement, the new leader would be unable to recover from a poisonous write. We therefore allow acceptors to change their mind and accept multiple values. Naturally, we must take precautions to ensure that CS2 still holds.

4.3.1 Progress Certificates and the Recovery Protocol

If some value v was chosen, then in order to maintain CS2 a new correct leader must not propose any value other than v. In order to determine whether some value was chosen, the new leader must therefore query the acceptors for their state. It can gather at most $a-f$ replies. We call the set of these replies a *progress certificate*. The progress certificate serves two purposes. First, it allows a new correct leader to determine whether some value v may have been chosen, in which case the leader proposes v. We say that a correct leader will only propose a value that the progress certificate *vouches* for—we will discuss in Section 4.3.2 how a progress certificate vouches for a value. Second, the progress certificate allows acceptors to determine the legitimacy of the value proposed by the leader, so that a faulty leader may not corrupt the state after some value was chosen. In order to serve the second purpose, we require the answers in the progress certificate to be signed.

A progress certificate PC must have the property that, if some value v was chosen, then PC only vouches for v (since v is the only proposal that maintains CS2). It must also have the property that it always vouches for at least one value, to ensure progress despite poisonous writes.

In the recovery protocol, the newly elected correct leader α first gathers a progress certificate by querying acceptors and receiving $a-f$ signed responses. Then, α decides which value to propose: If the progress certificate vouches for some value v, then α proposes v. Otherwise, α is free to propose any value. To propose its value, α follows the normal leader protocol, piggybacking the progress certificate alongside its proposal to justify its choice of value. The acceptors check that the new proposed value is vouched for by the progress certificate, thus ensuring that the new value does not endanger safety.

As in Paxos, acceptors who hear of the new leader (when the new leader gathers the progress certificate) promise to ignore messages with a lower proposal number (i.e., messages from former leaders). In order to prevent faulty proposers from displacing a correct leader, the leader election protocol provides a proof-of-leadership token to the new leader (typically, a collection of signed "election" messages).

4.3.2 Constructing Progress Certificates

A straightforward implementation of progress certificates would consist of the currently accepted value, signed, from $a-f$ acceptors. If these values are all different, then, clearly, no value was chosen: In this case, the progress certificate should vouch for any value since it is safe for the new leader to propose any value.

Unfortunately, this implementation falls short: A faulty new leader could use such a progress certificate *twice* to cause two different values to be chosen. Further, this can happen even if individual proposers only accept a given progress certificate 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 λ can gather a progress certificate establishing that no value has been chosen. With this certificate, λ can first sway f acceptors from the third group to accept "A" (by definition, "A" is now chosen), and then, using the same progress certificate, persuade the acceptors in the first and fourth group to change their value to "B"—"B" is now chosen. Clearly, this execution violates CS2.

```
101  leader.onElected(newnumber,proof):
102    // this function is called when leader-election picks a new regency
103    // proof is a piece of data that will sway leader-election at the
104    // other nodes.
105    pnumber := newnumber // no smaller than the previous pnumber
106    if (not leader for pnumber) then return
107    send (QUERY,pnumber,proof) to all acceptors
108    until get  (REP, signed(value,pnumber) ) from a-f acceptors
109    PC := the union of these replies
110    if PC vouches for (v',pnumber) then value := v'
111    onStart()
112
113  acceptors.onQuery(pn,proof): from leader
114    leader-election.consider(proof)
115    if (leader-election.getRegency() != pn) then:
116      return // ignore bad requests
117    send (REP, signed(value,pn)) to leader-election.getLeader()
118  acceptor.onPropose(value,pnumber,progcert): from proposer
119    if pnumber != leader-election.getRegency() then
120      return // only listen to current leader
121    if accepted (v,pn) and pn == pnumber then
122      return // only once per prop. number
123    if accepted (v,pn) and v != value and
124    progcert does not vouch for (value,pnumber) then
125      return // only change with progress certificate
126    accepted := (value,pnumber) // accepting
127    send (ACCEPTED,accepted) to all learners
```

Fig. 4. FaB recovery pseudocode.

We make three changes to prevent progress certificates from being used twice. First, we allow a proposer to propose a new value only once while it serves as a leader. Specifically, we tie progress certificates to a *proposal number*, whose value equals the number of times a new leader has been elected.

Second, we associate a proposal number to proposed values. Acceptors now accept a value for a given proposal number rather than just a value. Where before acceptors forwarded just the accepted value (to help learners decide, or in response to a leader's query), now they forward both the accepted value *and* its proposal number—hence, progress certificates now contain (value, proposal number) pairs.

Learners learn a value v if they see that $\lceil (a+3f+1)/2 \rceil$ acceptors accepted value v for the *same* proposal number. We say that value v is *chosen for* pn if $\lceil (a+f+1)/2 \rceil$ correct acceptors have accepted that value for proposal number pn. We say that value v is chosen if there is some proposal number pn so that v is chosen for pn.

Third, we change the conditions under which acceptors accept a value (Fig. 4). In addition to ignoring proposals with a proposal number lower than any they have seen (line 119), acceptors only accept one proposal for every proposal number (line 121) and they only change their accepted value if the progress certificate vouches for the new value and proposal number (lines 123 through 125).

We are now ready to define progress certificates concretely. A progress certificate contains signed replies (v_i, pn) from $a-f$ acceptors (Fig. 4, line 108). An acceptor's reply contains that acceptor's currently accepted value and the proposal number of the leader who requested the progress certificate.

Definition. *We say that a progress certificate* $((v_0, pn), \ldots, (v_{a-f}, pn))$ vouches *for value v at proposal number pn if there is no value $v_i \neq v$ that appears $\lceil (a-f+1)/2 \rceil$ times in the progress certificate.*

A consequence of this definition is that if some specific pair (v, pn) appears at least $\lceil (a-f+1)/2 \rceil$ times in the progress certificate, then the progress certificate vouches only for value v at proposal pn. If there is no such pair, then the progress certificate vouches for any value as long as its proposal number matches the one in the progress certificate. As we prove in the next section, progress certificates guarantee that if v is chosen for pn, then all progress certificates with a proposal number following pn will vouch for v and no other value.

Let us revisit the troublesome scenario of before in light of these changes. Suppose, without loss of generality, that the malicious leader λ gathers a progress certificate for proposal number 1 (λ is the second proposer to become leader). Because of the poisonous write, the progress certificate allows the leader to propose any new value. To have "A" chosen, λ sends a new proposal ("A," 1) together with the progress certificate first to the acceptors in the first group and then 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 value for the same proposal number.

Fortunately, this first step is also what prevents λ from using the progress certificate to sway the acceptors in the first group to accept "B." Because they have last accepted the pair ("A," 1), when λ presents to the acceptors in the first group the progress certificate for proposal number 1 for the second time, they will refuse it (line 121 of the protocol).

4.4 Correctness

We now prove that, for executions that are eventually synchronous, FaB Paxos solves consensus. Recall that a value v is chosen for proposal pn iff $\lceil (a+f+1)/2 \rceil$ correct acceptors accept v for proposal pn. As mentioned at the beginning of this section, we assume that $a > 5f$, $p > 3f$, and $l > 3f$.

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

Proof. To be chosen, a value must be accepted by a set of correct acceptors (by definition), and correct acceptors only accept values that are proposed (line 118). □

CS2. *Only a single value may be chosen.*

Proof. The theorem follows directly from the following two lemmas: □

Lemma 1. *For every proposal number pn, at most one value is chosen.*

Proof. Correct acceptors only accept one value per proposal number (line 121). In order for a value to be chosen for pn, the value must be accepted by at least a majority of the acceptors (by definition). Hence, at most one value is chosen per proposal number. □

Lemma 2. *If value v is chosen for proposal pn, then every progress certificate for proposal number $pn' > pn$ will vouch for v and no other value.*

Proof. Assume that value v is chosen for proposal pn; then, by definition, at least $c = \lceil (a + f + 1)/2 \rceil$ correct acceptors have accepted v for proposal pn. Let PC be a progress certificate for proposal number $pn' > pn$. All correct acceptors that accepted v for pn must have done so before accepting PC, since no correct acceptor would accept v for proposal pn if it had accepted PC with $pn' > pn$ (line 119 and the fact that the regency number never decreases). Consider the $a - f$ pairs contained in PC. Since these pairs are signed (line 117), they cannot have been manufactured by the leader; hence, at least $a - f + c - a = \lceil (a - f + 1)/2 \rceil$ of them must be signed by acceptors that accepted v for pn. By definition, then, PC vouches for v and no other value. □

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

Proof. Suppose that a correct learner learns value v for proposal pn. There are two ways for a learner to learn a value in FaB Paxos.

- $\lceil (a + 3f + 1)/2 \rceil$ acceptors reported having accepted v for proposal pn (line 25). At least $\lceil (a + f + 1)/2 \rceil$ of these acceptors are correct, so, by definition, v was chosen for pn.
- $f + 1$ other learners reported that v was chosen for pn (line 40). One of these learners is correct—so, by induction on the number of learners, it follows that v was indeed chosen for pn. □

We say that a value is *stable* if it is learned by $\lceil (l - f + 1)/2 \rceil$ correct learners.

Lemma 3. *Some value is eventually stable.*

Proof. The system is eventually synchronous and in these periods, leaders that do not create a stable value are eventually suspected by all correct proposers (line 13). In this situation, the leader election protocol elects a new leader. Byzantine learners or proposers cannot prevent the election: Even if the f faulty learners pretend to have learned a value, the remaining correct proposers form a quorum and, thus, can trigger an election (see Section 4.3). Since the number of proposers p is larger than f, eventually either some value is stable or a correct leader α is elected. In a period of synchrony, Byzantine proposers alone cannot trigger an election to replace a correct leader (see Section 4.3). We show that if α is correct then some value will be stable. The correct leader will gather a progress certificate (line 108) and propose a value to all the acceptors. By construction, all progress certificates vouch for at least one value—and correct acceptors will accept a value vouched by a progress certificate. Since α is correct, it will propose the same value to all acceptors and all $a - f$ correct acceptors will accept the proposed value. Given that $a > 3f$, $\lceil (a + f + 1)/2 \rceil \leq a - f$ and so, by definition, that value will be chosen. The end-to-end retransmission protocol (line 4) ensures that α will continue to resend its proposed value at least until it hears from $\lceil (l + f + 1)/2 \rceil$ learners that they have learned a value—that is, until the value is stable (line 8). □

CL1. *Some proposed value is eventually chosen.*

Proof. By Lemma 3, eventually some value is stable, i.e., $\lceil (l + f + 1)/2 \rceil > f$ correct learners have learned it. By CS3, a correct learner only learns a value after it is chosen. Therefore, the stable value is chosen. □

Our proof for CL1 only relies on the fact that the correct leader does not stop retransmission until a value is chosen. In practice, it is desirable for the leader to stop retransmission once it *is*. Since $l > 3f$, there are at least $\lceil (l + f + 1)/2 \rceil$ correct learners and, so, eventually all correct proposers will be satisfied (line 8) and the leader will stop retransmitting (line 4).

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

Proof. By Lemma 3, some value v is eventually stable, i.e., $\lceil (l - f + 1)/2 \rceil \geq f + 1$ correct learners eventually learn the value. Even if the leader is not retransmitting anymore, the remaining correct learners can determine the chosen value when they query their peers with the "pull" requests (lines 32 and 34-36) and receive $f + 1$ matching responses (line 40). So eventually, all correct learners learn the chosen value. □

5 Parameterized FaB Paxos

Previous Byzantine consensus protocols require $3f + 1$ processes and may complete in three communication steps when there is no failure; FaB Paxos requires $5f + 1$ processes and may complete in two communication steps despite up to f failures—the protocol uses the additional replication for speed. In this section, we explore scenarios that lie in between these two extremes: when fewer than $5f + 1$ processes are available and when it is not necessary to ensure two-step operation, even when *all* f processes fail.

We generalize FaB Paxos by decoupling replication for fault tolerance from replication for speed. The resulting protocol, Parameterized FaB Paxos (Fig. 5 spans the whole design space between a minimal number of processes (but no guarantee of two-step executions) and two-step protocols (that require more processes). This trade-off is expressed through the new parameter t ($0 \leq t \leq f$). Parameterized FaB Paxos requires $3f + 2t + 1$ processes, is safe despite up to f Byzantine failures, and all its executions are two-step in the common case despite up to t Byzantine

```
201  leader.onStart():
202    // proposing (PC is null unless recovering)
203    send (PROPOSE,value,number,PC) to all acceptors
204    until |Satisfied| >= ⌈(p+f+1)/2⌉
205
206  leader.onElected(newnumber,proof):
207    pnumber := newnumber // no smaller than previous pnumber
208    if (not leader for pnumber) then return
209    send (QUERY,pnumber,proof) to all acceptors
210    until get (REP, ⟨value,pnumber,commit_proof,j⟩) ) from
211      a − f acceptors
212    PC := the union of these replies
213    if ∃ v' s.t. vouches−for(PC, v',pnumber), then value := v'
214    onStart()
215
216  proposer.onLearned(): from learner l
217    Learned := Learned union {l}
218    if |Learned| >= ⌈(l+f+1)/2⌉ then
219       send (SATISFIED) to all proposers
220
221  proposer.onStart():
222    wait for timeout
223    if |Learned| < ⌈(l+f+1)/2⌉ then
224       suspect the leader
225
226  proposer.onSatisfied(): from proposer x
227    Satisfied := Satisfied ∪ {x}
228
229  acceptor.onPropose(value,pnumber,progcert): from leader
230    if pnumber != leader−election.getRegency() then
231      return // only listen to current leader
232    if accepted (v,pn) and ((pnumber<=pn) or ((v!=value)
233    and not vouches−for(progcert,value,pnumber))) then
234      return // only change with progress certificate
235    accepted := (value,number) // accepting
236    send (ACCEPTED,accepted) to all learners
237    // i is the number of this acceptor
238    send ⟨ACCEPTED,value,pnumber,i⟩_i to all acceptors
239
240  acceptor.onAccepted(value,pnumber,j):
241    if pnumber>tentative_commit_proof[j].pnumber then
242      tentative_commit_proof[j] := ⟨ACCEPTED,value_j,pnumber,j⟩_j
243    if valid(tentative_commit_proof,value,leader−election.
           getRegency()) then
244      commit_proof := tentative_commit_proof
245      send (COMMITPROOF,commit_proof) to all learners
246
247  acceptors.onQuery(pn,proof): from proposer
248    leader−election.consider(proof)
249    if (leader−election.getRegency() != pn) then
250      return // ignore bad requests
251    leader := leader−election.getLeader()
252    send (REP, ⟨accepted.value,pn,commit_proof,i⟩_i) to leader
253  learner.onAccepted(value,ponumber): from acceptor ac
254    accepted[ac] := (value,pnumber)
255    if there are ⌈(a+3f+1)/2⌉ acceptors x
256    such that accepted[x] == (value,pnumber) then
257      learn(value,pnumber) // learning
258
259  learner.onCommitProof(commit_proof): from acceptor ac
260    cp[ac] := commit_proof
261    (value,pnumber) := accepted[ac]
262    if there are ⌈(a+f+1)/2⌉ acceptors x
263    such that valid(cp[x],value,pnumber) then
264       learn(value,pnumber) // learning
265
266  learner.learn(value,pnumber):
267    learned := (value,pnumber) // learning
268    send (LEARNED) to all proposers
269
270  learner.onStart():
271    wait for timeout
272    while (not learned) send (PULL) to all learners
273
274  learner.onPull(): from learner ln
275    If this process learned some pair (value,pnumber) then
276      send (LEARNED,value,pnumber) to ln
277
278  learner.onLearned(value,pnumber): from learner ln
279    Learn[ln] := (value,pnumber)
280    if there are f + 1 learners x
281    such that learn[x] == (value,pnumber) then
282      learned := (value,pnumber)
283
284  valid(commit_proof,value,pnumber):
285    c := commit_proof
286    if there are ⌈(a+f+1)/2⌉ distinct values of x such that
287      (c[x].value == value) and (c[x].pnumber == pnumber)
288    then return true
289    else return false
290
291  vouches−for(PC,value,pnumber):
292    if there exist ⌈(a−f+1)/2⌉ x such that
293      all PC[x].value == d
294      and d != value
295    then return false
296    if there exists x, d!=value such that
297      valid(PC[x].commit_proof,d,pnumber)
298    then return false
299    return true
```

Fig. 5. Parameterized FaB (with recovery).

failures. We say that the protocol is *(t,2)-step*. FaB Paxos is just a special case of Parameterized FaB Paxos, with $t = f$.

Several choices of t and f may be available for a given number of machines. For example, if seven machines are available, an administrator can choose between tolerating two Byzantine failures to tolerate only two Byzantine failures and slow down after the first failure ($f = 2, t = 0$) or tolerate only one Byzantine failure but maintain two-step operation despite the failure ($f = 1, t = 1$).

The key observation behind this protocol is that FaB Paxos maintains safety even if $n > 5f + 1$ (provided that $n > 3f$). It is only liveness that is affected by having

fewer than $5f+1$ acceptors: Even a single crash may prevent the learners from learning (the predicate at line 25 of Fig. 2). In order to restore the liveness property even with $3f < n < 5f+1$, we merge a traditional BFT three-phase-commit [3] with FaB Paxos. While merging the two, we take special care to ensure that the two features never disagree as to which value should be decided. The Parameterized FaB Paxos code does not include any mention of the parameter t: If there are more than t failures, the two-step feature of Parameterized FaB Paxos may never be triggered because there are not enough correct nodes to send the required number of messages.

First, we modify acceptors so that, after receiving a proposal, they sign it (including the proposal number) and forward it to each other so each of them can collect a commit proof. A *commit proof* for value v at proposal number pn consists of $\lceil (a+f+1)/2 \rceil$ statements from different acceptors that accepted value v for proposal number pn (function "valid," line 284). The purpose of commit proofs is to give evidence as to which value was chosen. If there is a commit proof for value v at proposal pn, then no other value can possibly have been chosen for proposal pn. We include commit proofs in the progress certificates (line 252) so that newly elected leaders have all the necessary information when deciding which value to propose. The commit proofs are also forwarded to learners (line 245) to guarantee liveness when more than t acceptors fail.

Second, we modify learners so that they learn a value if enough acceptors have a commit proof for the same value and proposal number (line 262).

Finally, we redefine "chosen" and "progress certificate" to take commit proofs into account.

We now say that value v is *chosen for* proposal number pn if $\lceil (a+f+1)/2 \rceil$ correct acceptors have accepted v in proposal pn or if $\lceil (a+f+1)/2 \rceil$ acceptors have (or had) a commit proof for v and proposal number pn. Learners learn v when they know v has been chosen. The protocol ensures that only a single value may be chosen.

Progress certificates still consist of $a-f$ entries, but each entry now contains an additional element: either a commit proof or a signed statement saying that the corresponding acceptor has no commit proof. A progress certificate *vouches* for value v' at proposal number pn if all entries have proposal number pn, there is no value $d \neq v'$ contained $\lceil (a-f+1)/2 \rceil$ times in the progress certificate and the progress certificate does not contain a commit proof for any value $d \neq v'$ (function "vouches-for," line 291). The purpose of progress certificates is, as before, to allow learners to convince acceptors to change their accepted values.

These three modifications maintain the properties that at most one value can be chosen and that, if some value was chosen, then future progress certificates will vouch only for it. This ensures that the changes do not affect safety. Liveness is maintained despite f failures because there are at least $\lceil (a+f+1)/2 \rceil$ correct acceptors, so, if the leader is correct, then eventually all of them will have a commit proof, thus allowing the proposed value to be learned. The next section develops these points in more detail.

5.1 Correctness

The proof that Parameterized FaB implements consensus follows the same structure as that for FaB.

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

Proof. To be chosen, a value must be accepted by a set of correct acceptors (by definition) and correct acceptors only accept values that are proposed (line 229). □

CS2. *Only a single value may be chosen.*

Proof. This proof also follows a similar argument as the one in Section 4.4. We first consider values chosen for the same proposal number, then we show that once a value v is chosen, later proposals also propose v. Parameterized FaB uses a different notion of chosen, so we must show that a value, once chosen, remains so if no correct node accepts new values. □

Lemma 4. *If value v is chosen for proposal number pn, then it was accepted by $\lceil (a+f+1)/2 \rceil$ acceptors in proposal pn.*

Proof. The value can be chosen for two reasons according to the definition: either because $\lceil (a+f+1)/2 \rceil$ correct acceptors accepted it (in which case the lemma follows directly), or because $\lceil (a+f+1)/2 \rceil$ acceptors have a commit proof for v at pn. At least one of them is correct, and a commit proof includes answers from $\lceil (a+f+1)/2 \rceil$ acceptors who accepted v at pn (lines 243 and 286-289). □

Corollary 1. *For every proposal number p_n, at most one value is chosen.*

Proof. If two values were chosen, then the two sets of acceptors who accepted them intersect in at least one correct acceptor. Since correct acceptors only accept one value per proposal number (line 232), the two values must be identical. □

Corollary 2. *If v is chosen for proposal pn then v is chosen for proposal pn and no correct acceptor accepts a different value for proposals with a higher number than pn, then v is the only value that can be chosen for any proposal number higher than pn.*

Proof. Again, the two sets needed to choose distinct v and v' would intersect in at least a correct acceptor. Since, by assumption, these correct acceptors did not accept a different value after pn, $v = v'$. □

Lemma 5. *If v is chosen for pn then every progress certificate PC for a higher proposal number either vouches for no value or vouches for value v.*

Proof. Suppose that the value v is chosen for pn. The higher-numbered progress certificate PC will be generated in lines 209-212 by correct proposers. We show that all progress certificates for proposal numbers higher than pn that vouch for a value vouch for v (we will show later that, in fact, all progress certificates from correct proposers vouch for at least one value).

The value v can be chosen for pn for one of two reasons. In each case, the progress certificate can only vouch for v.

First, v could be chosen for pn because there is a set A of $\lceil (a+f+1)/2 \rceil$ correct acceptors that have accepted v

for proposal pn. The progress certificate for pn', PC, consists of answers from $n-f$ nodes (line 209). These answers are signed so each answer in a valid progress certificate come from a different node. Since acceptors only answer higher-numbered requests (line 249; regency numbers never decrease), all nodes in A that answered have done so after having accepted v in proposal pn. At most f acceptors may be faulty, so PC includes at least $\lceil (a-f+1)/2 \rceil$ answers from A. By definition, it follows that PC cannot vouch for any value other than v (lines 292-295).

Second, v could be chosen for pn because there is a set B of $\lceil (a+f+1)/2 \rceil$ acceptors that have a commit proof for v for proposal pn. Again, the progress certificate PC for pn' includes at least $\lceil (a-f+1)/2 \rceil$ answers from B. Up to f of these acceptors may be Byzantine and lie (pretending to never have seen v), so PC may contain as few as $\lceil (a-3f+1)/2 \rceil$ commit proofs for v. Since $a > 3f$, PC contains at least one commit proof for v, which by definition is sufficient to prevent PC from vouching for any value other than v (lines 296-297). □

Lemma 6. *If v is chosen for pn, then v is the only value that can be chosen for any proposal number higher than pn.*

Proof. In order for a different value v' to be chosen, a correct acceptor would have to accept a different value in a later proposal (Corollary 2). Correct acceptors only accept a new value v' if it is accompanied by a progress certificate that vouches for v' (lines 232-234). The previous lemma shows that no such progress certificate can be gathered.

Putting it all together, we can show that CS2 holds (by contradiction). Suppose that two distinct values, v and v' are chosen. By Corollary 1, they must have been chosen in distinct proposals pn and pn'. Without loss of generality, suppose $pn < pn'$. By Lemma 6, $v' = v$. □

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

Proof. Suppose that a correct learner learns value v after observing that v is chosen for pn. There are three ways for a learner to make that observation in Parameterized FaB.

- $\lceil (a+3f+1)/2 \rceil$ acceptors reported having accepted v for proposal pn (line 255). At least $\lceil (a+f+1)/2 \rceil$ of these acceptors are correct, so, by definition, v was chosen for pn.
- $\lceil (a+f+1)/2 \rceil$ acceptors reported a commit proof for v for proposal pn (lines 262-264). By definition, v was chosen for pn.
- $f+1$ other learners reported that v was chosen for pn (lines 280-282). One of these learners is correct—so, by induction on the number of learners, it follows that v was indeed chosen for pn. □

Lemma 7. *All valid progress certificates vouch for at least one value.*

Proof. The definition allows for three ways for a progress certificate PC to vouch for no value at all. We show that none can happen in our protocol.

First, PC could vouch for no value if there were two distinct values v and v', each containing $\lceil (a-f+1)/2 \rceil$ times in the PC. This is impossible because PC only contains $a-f$ entries in total (line 211).

Second, PC could vouch for no value if it contained two commit proofs for distinct values v and v'. Both commit proofs contains $\lceil (a+f+1)/2 \rceil$ identical entries (for v and v', respectively) from the same proposal (lines 286-287). These two sets intersects in a correct proposer, but correct proposers only accept one value per proposal number (line 232). Thus, it is not possible for PC to contain two commit proofs for distinct values.

Third, there could be some value v contained $\lceil (a-f+1)/2 \rceil$ times in the PC, and a commit proof for some different value v'. The commit proof includes values from $\lceil (a+f+1)/2 \rceil$ acceptors, and at least $\lceil (a-f+1)/2 \rceil$ of these are honest, so they would report the same value (v') in the PC. But $\lceil (a-f+1)/2 \rceil$ is a majority and there can be only one majority in the PC, so that scenario cannot happen. □

Recall that a value is stable if it is learned by $\lceil (l-f+1)/2 \rceil$ correct learners. We use Lemma 3, which shows that some value is eventually stable, to prove CL1 and CL2.

CL1. *Some proposed value is eventually chosen.*

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

Proof. The proofs for CL1 and CL2 are unchanged. They still hold because although the parameterized protocol makes it easier for a value to be chosen, it still has the property that the leader will resend its value until it knows that the value is stable (lines 203-204, 216-219). A value that is stable is chosen (ensuring CL1) and it has been learned by at least $\lceil (l-f+1)/2 \rceil$ correct learners (ensuring CL2 because of the pull subprotocol on lines 270-282). □

6 The Lower Bound

Parameterized FaB Paxos requires $3f+2t+1$ acceptors to tolerate f Byzantine failures and be two-step despite t failures. We show that this is the optimal number of processes for parameterized two-step consensus. Our proof does not distinguish between proposers, acceptors, and learners because doing so would restrict the proof to Paxos-like protocols.

The proof proceeds by constructing two executions that are indistinguishable, although they should learn different values. We now define these notions precisely.

We consider a system of n processes that communicate through a fully connected network. Processes execute sequences of events, which can be of three types: *local*, *send*, and *deliver*. We call the sequence of events executed by a process its *local history*.

An execution of the protocol proceeds in asynchronous rounds. In a round, each correct process 1) sends a message to every other process, 2) waits until it receives a (possibly empty) message sent in that round from $n-f$ distinct processes (ignoring any extra messages), and 3) performs a

(possibly empty) sequence of local events. We say that the process takes a *step* in each round. During an execution, the system goes through a series of *configurations*, where a configuration C is an n-vector that stores the state of every process. We also talk about the *state* of a set of processes, by which we mean a vector that stores the state of the processes in the set.

This proof depends crucially on the notion of indistinguishability. The notions of *view* and *similarity* help us capture this notion precisely.

Definition. *Given an execution ρ and a process p_i, the view of p_i in ρ, denoted by $\rho|p_i$, is the local history of p_i together with the state of p_i in the initial configuration of ρ.*

Definition. *Let ρ_1 and ρ_2 be two executions, and let p_i be a process which is correct in ρ_1 and ρ_2. Execution ρ_1 is* similar *to execution ρ_2 with respect to p_i, denoted as $\rho_1 \overset{p_i}{\sim} \rho_2$, if $\rho_1|p_i = \rho_2|p_i$.*

If an execution ρ results in all correct processes learning a value v, we say that v is the *consensus value* of ρ, which we denote $c(\rho)$. For the remainder of this section, we only consider executions that result in all correct processes learning a value.

Lemma 8. *Let ρ_1 and ρ_2 be two executions, and let p_i be a process which is correct in ρ_1 and ρ_2. If $\rho_1 \overset{p_i}{\sim} \rho_2$, then $c(\rho_1) = c(\rho_2)$.*

Proof. The correct process cannot distinguish between ρ_1 and ρ_2, so it will learn the same value in both executions. Consensus requires that all correct learners learn the consensus value, so $c(\rho_1) = c(\rho_2)$. □

Definition. *Let $\mathcal{F}$ be a subset of the processes in the system. An execution ρ is $\mathcal{F}$-*silent *if, in ρ, no process outside $\mathcal{F}$ delivers a message from a process in $\mathcal{F}$.*

Definition. *Let a* two-step execution *be an execution in which all correct processes learn by the end of the second round. A consensus protocol is* (*t*,2)-step *if it can tolerate f Byzantine failures and if, for every initial configuration I and every set $\mathcal{F}$ of at most t processes ($t \leq f$), there exists a two-step execution of the protocol from I that is $\mathcal{F}$-*silent*. If the protocol is (f,2)-step, then we simply say that it is* two-step.

Definition. *Given a (t,2)-step consensus protocol, an initial configuration I is (t,2)-step bivalent if there exist two disjoint sets of processes $\mathcal{F}_0$ and $\mathcal{F}_1$, ($|\mathcal{F}_0| \leq t$ and $|\mathcal{F}_1| \leq t$) an $\mathcal{F}_0$-*silent *two-step execution ρ_0 and an $\mathcal{F}_1$-*silent *two-step execution ρ_1 such that $c(\rho_0) = 0$ and $c(\rho_1) = 1$.*

Lemma 9. *For every (t,2)-step consensus protocol with $n > 2f$, there exists a (t,2)-step bivalent initial configuration.*

Proof. Consider a (*t*,2)-step consensus protocol P. For each i, $0 \leq i \leq n$, let I^i be the initial configuration in which the first i processes propose 1 and the remaining processes propose 0. By the definition of (*t*,2)-step, for every I^i and for all $\mathcal{F}$ such that $|\mathcal{F}| \leq \sqcup$ there exists at least one $\mathcal{F}$-silent two-step execution ρ^i of P. By property CS1 of consensus, $c(\rho^0) = 0$ and $c(\rho^n) = 1$. Consider now $\mathcal{F}_0 = \{p_j : 1 \leq j \leq t\}$. There must exist two neighbor configurations I^i and I^{i+1} and two $\mathcal{F}_0$-silent two-step executions ρ^i and ρ^{i+1} such that $c(\rho^i) \neq c(\rho^{i+i})$ and ρ^{i+1} is the lowest-numbered execution with consensus value 1.

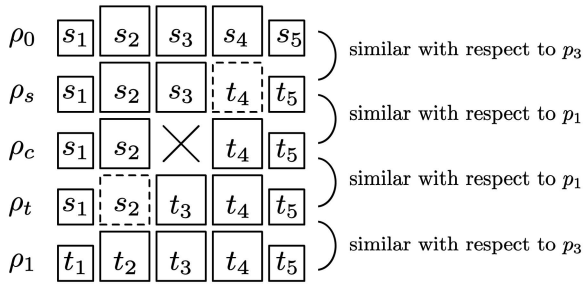

Fig. 6. Contradiction sketch: The figure represents a system with too few $(3f + 2t)$ processes. Each row represents an execution, and the boxes represent sets of processes. Dotted boxes contain Byzantine nodes. The first execution (ρ_0) learns 0, and the last learns 1. Each execution is similar to the next, leading to the contradiction.

Note that $i \geq t$, since both ρ^i and ρ^{i+1} are $\mathcal{F}_0$-silent and the consensus value they reach cannot depend on the value proposed by the silent processes in $\mathcal{F}_0$. We claim that one of I^i and I^{i+1} is (*t*,2)-step bivalent. To prove our claim, we set $x = min(i + t, n)$ and define $\mathcal{F}_1$ as the set $\{p_j : x + 1 - t \leq j \leq x\}$. Note that, by construction, $\mathcal{F}_0$ and $\mathcal{F}_1$ are disjoint and $(i + 1) \in \mathcal{F}_1$. By the definition of C, there must in turn exist two new two-step executions π^i and π^{i+1} that are $\mathcal{F}_1$-silent. The only difference between configurations I^i and I^{i+1} is the value proposed by p_{i+1}, which is silent in π^i and π^{i+1}, since it belongs to $\mathcal{F}_1$. Hence, all processes outside of $\mathcal{F}_1$ (at least one of which is correct) have the same view in π^i and π^{i+1}, and $c(\pi^i) = c(\pi^{i+1})$. Since $c(\rho^i) \neq c(\rho^{i+1})$ and $c(\pi^i) = c(\pi^{i+1})$, either I^i or I^{i+1} has two two-step executions that lead to different consensus values. This is the definition of a (*t*,2)-step bivalent configuration. □

Fig. 6 shows a sketch of the idea at the core of the proof: With only $3f + 2t$ acceptors, we can construct two executions (ρ_0 and ρ_1) that are indistinguishable, even though they learn different values.

Theorem 1. *Any (t,2)-step Byzantine fault-tolerant consensus protocol requires at least $3f + 2t + 1$ processes.*

Proof. By contradiction. Suppose there exists a (*t*,2)-step fault-tolerant consensus protocol P that 1) tolerates up to f Byzantine faults, 2) is two-step despite t failures, and 3) requires only $3f + 2t$ processes. We partition the processes in five sets, $p_1 \ldots p_5$.

By Lemma 9, there exist a (*t*,2)-step bivalent configuration I_b and two two-step executions ρ_0 and ρ_1, respectively $\mathcal{F}_0$-silent and $\mathcal{F}_1$-silent, such that $c(\rho_0) = 0$ and $c(\rho_1) = 1$. We name the sets of processes so that $\mathcal{F}_0 = p_5$ and $\mathcal{F}_1 = p_1$ (so p_1 and p_5 have size t). The remaining sets have a size f.

We focus on the state of $p_1, \ldots, p_5$ at the end of the first round, where the state of p_i is a set of local states, one for each process in p_i. In particular, let $\mathbf{s_i}$ and $\mathbf{t_i}$ denote the state of p_i at the end of the first round of ρ_0 and ρ_1, respectively. p_i has state $\mathbf{s_i}$ (respectively, $\mathbf{t_i}$) at the end of any execution that produces for its nodes the same view as ρ_0(respectively, ρ_1). It is possible for some processes to be in an **s** state at the end of the first round while at the same time others are in a **t** state. Consider now three new (not necessarily two-step) executions of P, ρ_s, ρ_t, and ρ_c,

that at the end of their first round have p_1 and p_2 in their $\mathbf{s}$ states and p_4 and p_5 in their $\mathbf{t}$ states. The state of p_3 is different in the three executions: In ρ_s, p_3 is in state $\mathbf{s}_3$; in ρ_t, p_3 is in state $\mathbf{t}_3$; and in ρ_c, p_3 crashes at the end of the first round. Otherwise, the three executions are very much alike: All three executions are p_3-silent from the second round on—in ρ_c because p_3 has crashed, in ρ_s and ρ_t because all processes in p_3 are slow. Further, all processes other than those in p_3 send and deliver the same messages in the same order in all three executions and all three executions enter a period of synchrony from the second round on so that in each, execution consensus must terminate and some value must be learned. We consider three scenarios, one for each execution.

1. ρ_s *scenario.* In this scenario, the f nodes in p_4 are Byzantine: They follow the protocol correctly in their messages to all processes but those in p_3. The messages that nodes in p_4 send to p_3 in round two are consistent with p_4 being in state $\mathbf{s}_4$, rather than $\mathbf{t}_4$. Further, in the second round of ρ_s the messages from p_5 to p_3 are the last to reach p_3 (and are therefore not delivered by p_3), and all other messages are delivered by p_3 in the same order as in ρ_0. The view of p_3 at the end of the second round of ρ_s is the same as in the second round of ρ_0; hence nodes in p_3 learn 0 at the end of the second round of ρ_s (it must learn then because ρ_0 is two-step). Since nodes in p_3 are correct and for each node $p \in p_3$ $\rho_s \overset{p}{\sim} \rho_0$, then $c(\rho_s) = c(\rho_0)$ and all correct processes in ρ_s eventually learn 0.
2. ρ_t scenario: In this scenario, the f nodes in p_2 are Byzantine: They follow the protocol correctly in their messages to all processes but those in p_3. In particular, the messages that nodes in p_2 send to p_3 in round two are consistent with p_2 being in state $\mathbf{t}_2$, rather than bfs_2. Further, in the second round of ρ_t the messages from p_1 to p_3 are the last to reach p_3 (and are therefore not delivered by p_3), and all other messages are delivered by p_3 in the same order as in ρ_1. The view of p_3 at the end of the second round of ρ_t is the same as in the second round of ρ_1; hence, nodes in p_3 learn 1 at the end of the second round of ρ_t. Since nodes in p_3 are correct and, for each node, $p \in p_3$, $\rho_t \overset{p}{\sim} \rho_1$, then $c(\rho_t) = c(\rho_1)$ and all correct processes in ρ_t eventually learn 1.
3. ρ_c scenario. In this scenario, the f nodes in p_3 have crashed, and all other processes are correct. Since ρ_c is synchronous from round 2 on, every correct process must eventually learn some value.

Consider now a process p in p_1 that is correct in ρ_s, ρ_t, and ρ_c. By construction, $\rho_c \overset{p}{\sim} \rho_t$ and, therefore, $c(\rho_c) = c(\rho_t) = c(\rho_1) = 1$. However, again by construction, $\rho_c \overset{p}{\sim} \rho_s$ and, therefore, $c(\rho_c) = c(\rho_s) = c(\rho_0) = 0$. Hence, p in ρ_c must learn both 0 and 1: This contradicts CS2 and CS3 of consensus, which together imply that a correct learner may learn only a single value. □

7 State Machine Replication

Fast consensus translates directly into fast state machine replication: In general, state machine replication requires one fewer round with FaB Paxos than with traditional three-round Byzantine consensus protocols.

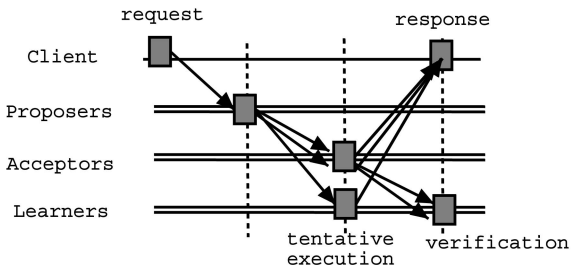

Fig. 7. FaB state machine with tentative execution.

A straightforward implementation of Byzantine state machine replication on top of FaB Paxos requires only four rounds of communication—one for the clients to send requests to the proposers; two (rather than the traditional three) for the learners to learn the order in which requests are to be executed, and a final one, after the learners have executed the request, to send the response to the appropriate clients. FaB can accommodate existing leader election protocols (e.g., [3]).

The number of rounds of communication can be reduced down to three using *tentative execution* [3], [9], an optimization used by Castro and Liskov for their PBFT protocol that applies equally well to FaB Paxos. As shown in Fig. 7, learners tentatively execute clients' requests as supplied by the leader before consensus is reached. The acceptors send to both clients and learners the information required to determine the consensus value, so clients and learners can, at the same time, determine whether their trust in the leader was well placed. In case of conflict, tentative executions are rolled back and the requests are eventually reexecuted in the correct order.

FaB Paxos loses its edge, however, in the special case of read-only requests that are not concurrent with any read-write request. In this case, a second optimization proposed by Castro and Liskov allows both PBFT and FaB Paxos to service these requests using just two rounds.

The next section shows further optimizations that reduce the number of learners and allow nodes to recover.

8 Optimizations

8.1 $2f + 1$ Learners

Parameterized FaB Paxos (and, consequently, FaB Paxos, its instantiation for $t = f$) requires $3f + 1$ learners. We show how to reduce the number of learners to $2f + 1$ without delaying consensus. This optimization requires some communication and the use of signatures in the common case, but still reaches consensus in two communication steps in the common case.

In order to ensure that all correct learners eventually learn, Parameterized FaB Paxos uses two techniques. First, the retransmission part of the protocol ensures that $\lceil (l + f + 1)/2 \rceil$ learners eventually learn the consensus value (line 204) and allows the remaining correct learners to pull the decided value from their up-to-date peers (lines 266-282).

To adapt the protocol to an environment with only $2f + 1$ learners, we first modify retransmission so that proposers are content with $f + 1$ acknowledgments from

learners—retransmission may now stop when only a single correct learner knows the correct response.

Second, we have to modify the "pull" mechanism because now a single correct learner must be able to convince other learners that its reply is correct. We, therefore, strengthen the condition under which we call a value stable (line 204) by adding information in the acknowledgments sent by the learners. In addition to the client's request and reply obtained by executing that request, acknowledgments must now also contain $f + 1$ signatures from distinct learners that verify the same reply.

After learning a value, learners now sign their acknowledgment and send that signature to all learners, expecting to eventually receive $f + 1$ signatures that verify their acknowledgment. Since there are $f + 1$ correct learners, each is guaranteed to be able to eventually gather an acknowledgment with $f + 1$ signatures 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 subprotocol: Learners query each other, and eventually all correct learners receive the valid acknowledgment and learn the decided value. This exchange of signatures takes an extra communication step, but this step is not in the critical path: It occurs *after* learners have learned the value.

The additional messages are also not in the critical path when this consensus protocol is used to implement a replicated state machine: The learners can execute the client's operation immediately when learning the operation, and can send the result to the client without waiting for the $f + 1$ signatures. Clients can already distinguish between correct and incorrect replies since only correct replies are vouched for by $f + 1$ learners.

8.2 Rejoin

By allowing repaired servers (for example, a crashed node that was rebooted) to rejoin, the system can continue to operate as long as at all times no more than f servers are either faulty or rejoining. The rejoin protocol must restore the replica's state, and as such it is different depending on the role that the replica plays.

The only state in proposers is the identity of the current leader. 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 instance of consensus d, and it then ignores all instances until $d + k$ (k is the number of instances of consensus that may run in parallel). Once the system moves on to instance $d + k$, the acceptor has completed its rejoin.

The state of the learners consists of the ordered list of operations. A rejoining learner, therefore, queries other learners for that list. It accepts answers that are vouched by $f + 1$ learners (either because $f + 1$ learners gave the same answer, or in the case of $2f + 1$ Parameterized FaB a single learner can show $f + 1$ signatures with its answer). Checkpoints could be used for faster state transfer as has been done before [3], [12].

9 Conclusion

FaB Paxos is the first Byzantine consensus protocol to achieve consensus in just two communication steps in the common case. This protocol is optimal in that it uses the minimal number of steps for consensus, and it uses the minimal number of processes to ensure two-step operation in the common case. Additionally, FaB Paxos in the common case does not require expensive digital signatures.

The price for common-case two-step termination is a higher number of acceptors than in previous Byzantine consensus protocols. These additional acceptors are precisely what allows a newly elected leader in FaB Paxos to determine, using progress certificates, whether or not a value had already been chosen—a key property to guarantee the safety of FaB Paxos in the presence of failures.

In traditional state machine architectures, the cost of this additional replication would make FaB Paxos unattractive for all but the applications most committed to reducing latency. However, the number of additional acceptors is relatively modest when the goal is to tolerate a small number of faults. In the state machine architecture that we have recently proposed, where acceptors are significantly cheaper to implement [23], the design point occupied by FaB Paxos becomes much more intriguing.

Even though $5f + 1$ acceptors is the lower bound for two-step termination, it is possible to sometimes complete in two communication steps even with fewer acceptors. Parameterized FaB Paxos decouples fault tolerance from two-step termination by spanning the design space between a Byzantine consensus protocol with the minimal number of servers (but that only guarantees two-step execution when there are no faults) to the full FaB protocol in which all common case executions are two-step executions. Parameterized FaB requires $3f + 2t + 1$ servers to tolerate f Byzantine failures and completes in two communication steps in the common case when there are at most t failures (we say that it is (t,2)-step). We have seen that this is the minimal number of servers with which it is possible to implement (t,2)-step consensus.

Acknowledgments

The authors would like to acknowledge the anonymous referees for their insightful suggestions toward improving the presentation of the paper. Moreover, this work was supported in part by US National Science Foundation CyberTrust award 0430510, an Alfred P. Sloan Fellowship, and a grant from the Texas Advanced Technology Program.

References

[1] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui, "Reconstructing Paxos," *SIGACT News*, vol. 34, no. 2, pp. 42-57, 2003.

[2] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Symp. Operating Systems Design and Implementation (OSDI)*, 1999.

[3] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Computer Systems*, vol. 20, no. 4, pp. 398-461, 2002.

[4] P. Dutta, R. Guerraoui, and M. Vukolić, "Best-Case Complexity of Asynchronous Byzantine Consensus," Technical Report EPFL/IC/200499, École Polytechnique Fédérale de Lausanne, Feb. 2005.

[5] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374-382, 1985.
[6] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and Efficient Oracle-Based Consensus Protocols for Asynchronous Byzantine Systems," *IEEE Trans. Dependable and Secure Computing,* vol. 2, no. 1, pp. 46-56, Jan. 2005.
[7] M. Hurfin and M. Raynal, "A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector," *Distributed Computing,* vol. 12, no. 4, pp. 209-223, Sept. 1999.
[8] I. Keidar and S. Rajsbaum, "On the Cost of Fault-Tolerant Consensus when There Are No Faults," Technical Report MIT-LCS-TR-821, Massachusetts Inst. of Technology, 2001.
[9] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using Optimistic Atomic Broadcast in Transaction Processing Systems," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 4, pp. 1018-1032, July/Aug. 2003.
[10] K. Kursawe, "Optimistic Byzantine Agreement," *Proc. 21st IEEE Symp. Reliable Distributed Systems,* pp. 262-267, Oct. 2002.
[11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.
[12] L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems,* vol. 16, no. 2, pp. 133-169, 1998.
[13] L. Lamport, "Lower Bounds for Asynchronous Consensus," *Proc. Int'l Workshop Future Directions in Distributed Computing,* pp. 22-23, June 2002.
[14] L. Lamport, "Lower Bounds for Asynchronous Consensus," Technical Report MSR-TR-2004-72, Microsoft Research, July 2004.
[15] L. Lamport, "Fast Paxos," Technical Report MSR-TR-2005-112, Microsoft Research, 2005.
[16] L. Lamport and M. Fischer, "Byzantine Generals and Transaction Commit Protocols," Technical Report 62, SRI Int'l, 1982.
[17] D. Malkhi and M. Reiter, "Byzantine Quorum Systems," *Distributed Computing,* vol. 11, no. 4, pp. 203-213, 1998.
[18] J.-P. Martin and L. Alvisi, "Fast Byzantine Paxos," Technical Report TR-04-07, Department of Computer Sciences, Univ. of Texas at Austin, Feb. 2004.
[19] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine Storage," *Proc. Sixth Int'l Conf. Distributed Computing (DISC),* pp. 311-325, Oct. 2002.
[20] R. Rodrigues, M. Castro, and B. Liskov, "BASE: Using Abstraction to Improve Fault Tolerance," *Proc. 18th ACM Symp. Operating Systems Principles,* pp. 15-28, Oct. 2001.
[21] A. Schiper, "Early Consensus in an Asynchronous System with a Weak Failure Detector," *Distributed Computing,* vol. 10, no. 3, pp. 149-157, Apr. 1997.
[22] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys,* vol. 22, no. 4, pp. 299-319, Sept. 1990.
[23] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services," *Proc. 19th ACM Symp. Operating Systems Principles,* pp. 253-267, Oct. 2003.

Jean-Philippe Martin received the BS degree in computer sciences from the Swiss Federal Institute of Technology (EPFL) and also has an MS degree. He is a PhD candidate in the Department of Computer Sciences at the University of Texas at Austin. His main research interests are trustworthy systems, Byzantine fault tolerance, and cooperative systems.

Lorenzo Alvisi received the Laurea degree (summa cum laude) in physics (1987) from the University of Bologna, Italy, and the MS (1994) and PhD (1996) degrees in computer science from Cornell University. He is an associate professor and faculty fellow in the Department of Computer Sciences at the University of Texas at Austin. His primary research interests are in dependable distributed computing. Dr. Alvisi is the recipient of an Alfred P. Sloan Research Fellowship, an IBM Faculty Partnership award, and a National Science Foundation CAREER award. He is a senior member of the IEEE.