Fast Byzantine Consensus

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

Abstract

We present the first consensus 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 step, and number of processes for 2-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.

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 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 [11], 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 [10, 19], 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 the system is in a period of synchrony. 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 [8]. 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 proactively recover

This work was supported in part by NSF CyberTrust award 0430510, an Alfred P. Sloan Fellowhip and a grant from the Texas Advanced Technology Program.

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

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 [20].

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

In this paper we make two contributions. First, we prove that any 2-step Byzantine consensus protocol needs at least $5f+1$ processes to tolerate f Byzantine faults. Second, we show that this lower bound is tight by presenting a 2-step f-tolerant Byzantine consensus protocol—Fast Byzantine (or *FaB*) Paxos—that uses $5f+1$ acceptors. In the common case, FaB Paxos requires no expensive digital signature operation. More broadly, we show that FaB Paxos requires $3f+2t+1$ acceptors to achieve 2-step consensus despite $t \leq f$ Byzantine acceptors.

Since building a replicated state machine from consensus adds a single communication step, 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 [2] 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 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]. However, an architecture for Byzantine fault tolerant state machine replication that physically separates agreement from execution [20] makes this tradeoff 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 acceptors 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, in Section 3, give a formal description of our system model. In Section 4 we prove the lower bound on the number of processes required by 2-step Byzantine consensus. We present f-tolerant FaB Paxos in Section 5, and we show in Section 6 how to generalize FaB Paxos to tolerate $t \leq f$ Byzantine acceptors using $3f+2t+1$ acceptors. In Section 7 we discuss the FaB replicated state machine 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 [11] 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 earlier protocols that are closest to FaB Paxos are the FastPaxos protocol by Boichat and colleagues [1], and Kursawe's Optimistic asynchronous Byzantine agreement [9]. 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. There is a subtler difference in 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 somewhat weaker than these, in that during gracious executions processes can fail, provided that the leader does not fail. Also, FastPaxos does not rely on eventual synchrony but on an eventual leader oracle; however, since we only use eventual synchrony for leader election, the difference is superficial.

In contrast to FastPaxos, Kursawe's elegant optimistic protocol assumes the same Byzantine failure model that

[2] No protocol can take fewer than two rounds to reach Byzantine consensus. In fact, in a synchronous system where one process may crash, all consensus protocols must take at least two rounds [5].

[3] Even with the optimization of tentative execution.

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. The SC protocol [18] achieves this goal when the failure detectors make no mistake and the coordinator process does not crash. The later FC protocol [7] achieves a similar result even in executions where there are crashes. FaB Paxos differs from these protocols because it can tolerate unreliable links and Byzantine failures.

In his paper on lower bounds for asynchronous consensus [13], 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 conjecture does not technically hold in the corner case where no learner can fail[4]. Dutta, Guerraoui and Vukolić have recently derived a comprehensive proof of Lamport's original conjecture 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 [12], we describe the behavior of FaB Paxos in terms of the actions performed by three classes of agents: 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. Each class may contain up to 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. Unlike [13], we allow learners to fail in a Byzantine manner.

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, and we assume that all proposers and acceptors know all public keys and 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 [6], we ensure safety at all times and only guarantee liveness during periods of synchrony.

4 The Lower Bound

The FaB Paxos protocol requires $5f+1$ acceptors, so at least $5f+1$ processes. We show that this is the optimal number of processes for 2-step consensus. Our proof does not distinguish between proposers, acceptors and learners because doing so would restrict the proof to Paxos-like protocols.

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 (i) sends a message to every other process, (ii) waits until it receives a (possibly empty) message sent in that round from $n-f$ distinct processes (ignoring any extra messages), and (iii) 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.

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

[4]The counterexample can be found in our technical report [15].

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 1. *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 $c(\rho_1) = c(\rho_2)$.*

Definition Let F be a subset of the processes in the system. An execution ρ is F-silent if in ρ no process outside F delivers a message from a process in F.

Definition Let a *2-step execution* be an execution in which all correct processes learn by the end of the second round. A consensus protocol is *2-step* if for every initial configuration I and every set F of at most f processes, there exists a 2-step execution of the protocol from I that is F-silent.

Definition Given a 2-step consensus protocol, an initial configuration I is *2-step bivalent* if there exist two disjoint sets of processes F_0 and F_1, ($|F_0| \leq f$ and $|F_1| \leq f$) an F_0-silent 2-step execution ρ_0 and an F_1-silent 2-step execution ρ_1 such that $c(\rho_0) = 0$ and $c(\rho_1) = 1$.

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

Proof. Consider a 2-step consensus protocol C. 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 2-step, for every I^i and for all F such that $|F| \leq f$ there exists at least one F-silent 2-step execution ρ^i of P. By property CS1 of consensus, $c(\rho^0) = 0$ and $c(\rho^n) = 1$. Consider now $F_0 = \{p_j : 1 \leq j \leq f\}$. There must exist two neighbor configurations I^i and I^{i+1} and two F_0-silent 2-step executions ρ^i and ρ^{i+1} where the value that is learned flips for the first time from 0 to 1. Note that $i \geq f$, since both ρ^i and ρ^{i+1} are F_0-silent and the consensus value they reach cannot depend on the value proposed by the silent processes in F_0. We claim that one of I^i and I^{i+1} is 2-step bivalent. To prove our claim, we set $x = min(i+f, n)$ and define F_1 as the set $\{p_j : x+1-f \leq j \leq x\}$. Note that, by construction, F_0 and F_1 are disjoint. By the definition of C, there must in turn exist two new 2-step executions π^i and π^{i+1} that are 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 F_1. Hence, all processes outside of F_1 (at least one of which is correct) have the same view in π^i and π^{i+1}, and $c(\pi^i) = c(\pi^{i+1})$.

To summarize, we have shown that in one of I^i and I^{i+1} there exist two 2-step executions that lead to different consensus values for two disjoint silent sets F_0 and F_1—that is, either I^i or I^{i+1} is 2-step bivalent. □

Theorem 1. *Any 2-step Byzantine fault-tolerant consensus protocol requires at least $5f+1$ processes.*

Proof. We prove the theorem by contradiction, supposing there exists a 2-step fault-tolerant consensus protocol P that tolerates up to f Byzantine faults and requires only $5f$ processes. We partition the processes in five sets of size f. For simplicity and without loss of generality, for the remaining of this proof we assume that $f = 1$ and that our system is comprised of five processes, p_1 through p_5. If $f > 1$ and each set contains more than one process, the following discussion must be modified so that in each execution all the processes in a set receive the same set of messages, and, if they fail, they do so in the same way and at the same time.

By Lemma 2 there exist a 2-step bivalent configuration I_b and two 2-step executions ρ_0 and ρ_1, respectively F_0-silent and F_1-silent, such that $c(\rho_0) = 0$ and $c(\rho_1) = 1$. Without loss of generality, assume $F_0 = \{p_5\}$ and $F_1 = \{p_1\}$.

We focus on the state of $p_1, \ldots, p_5$ at the end of the first round. In particular, let s_i and t_i denote the state of p_i at the end of the first round of ρ_0 and ρ_1, respectively. Process p_i will be in state s_i (respectively, t_i) at the end of any execution that produces for it 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 2-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 s states and p_4 and p_5 in their t states. The state of p_3 is different in the tree executions: in ρ_s, p_3 is in state s_3; in ρ_t, p_3 is in state 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 p_3 is slow. Further, all processes other than p_3 send and deliver the same messages in the same order in all three executions, and all three execution 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.

ρ_s scenario: In this scenario, p_4 is Byzantine: it follows the protocol correctly in its messages to all processes but p_3. In particular, the message p_4 sends to p_3 in round two is consistent with p_4 being in state s_4, rather than t_4. Further, in the second round of ρ_s the message from p_5 to p_3 is the last to reach p_3 (and is 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 p_3 learns 0 at the end of the second round of ρ_s. Since p_3 is correct and $\rho_s \overset{p_3}{\sim} \rho_0$, then $c(\rho_s) = c(\rho_0)$ and all correct processes in ρ_s eventually learn 0.

ρ_t **scenario:** In this scenario, p_2 is Byzantine: it follows the protocol correctly in its messages to all processes but p_3. In particular, the message p_2 sends to p_3 in round two is consistent with p_2 being in state t_2, rather than s_2. Further, in the second round of ρ_t the message from p_1 to p_3 is the last to reach p_3 (and is 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 p_3 learns 1 at the end of the second round of ρ_t. Since p_3 is correct and $\rho_t \overset{p_3}{\sim} \rho_1$, then $c(\rho_t) = c(\rho_1)$ and all correct processes in ρ_t eventually learn 1.

ρ_c **scenario:** In this scenario, p_3 has crashed, and all other processes are correct. Since ρ_c is synchronous from round two on, every correct process must eventually learn some value.

Consider now a process (e.g. p_1) which is correct in ρ_s, ρ_t, and ρ_c. By construction, $\rho_c \overset{p_1}{\sim} \rho_t$, and therefore $c(\rho_c) = c(\rho_t) = 1$. However, again by construction, $\rho_c \overset{p_1}{\sim} \rho_s$, and therefore $c(\rho_c) = c(\rho_s) = 0$. Hence, p_1 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. □

5 Fast Byzantine Consensus

We now present FaB Paxos, a 2-step Byzantine fault-tolerant consensus protocol that requires $5f+1$ processes, matching the lower bound of Theorem 1. 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.

5.1 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. Figure 1 shows the protocol's pseudocode. The `number` variable (proposal number) indicates which process is the leader; in the common case it 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 24). FaB avoids digital signatures in the common case because they are computationally expensive. Adding signatures would not reduce the number of communication steps nor the number of servers since FaB is already optimal in these two measures.

Correctness We defer the full correctness proof for FaB until we have discussed the recovery protocol in Section 5.4—in the following we give an intuition of why the protocol is safe in the common case.

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 only learn 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.

Proving liveness in the common case is also straightforward—the detailed proof for the common case can be found in [15].

5.2 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 (line 27). 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 satisfied acknowledgments (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 re-

[5] We do not show the election protocol, because existing leader election

```
1   leader.onStart():
2     // proposing (PC is null unless recovering)
3     send (PROPOSE, value, number, 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      suspect the leader
15
16  proposer.onSatisfied(): from proposer x
17    Satisfied := Satisfied ∪ {x}
18
19  acceptor.onPropose(value, number, progcert): from leader
20    if not already accepted then
21      accepted := (value, number) // accepting
22    send (ACCEPTED, accepted) to all learners
23  learner.onAccepted(value, number): from acceptor ac
24    accepted[ac] := (value, number)
25    if there are ⌈a+3f+1)/2⌉ acceptors x
26    such that accepted[x] == (value, number) then
27      learned := (value, number) // 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 I learned some pair (value, number) then
36      send (LEARNED, value, number) to ln
37
38  learner.onLearned(value, number): from learner ln
39    Learn[ln] := (value, number)
40    if there are f+1 x
41    such that learn[x] == (value, number) then
42      learned := (value, number) // learning
```

Figure 1. FaB pseudocode (excluding recovery)

transmission 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. The pull protocol in lines 29-41 ensures that the remaining correct learners will eventually learn from their peers.

Correctness The proofs of CS1, CS2, and CS3 for the common case apply, unchanged, in this weaker network model. The liveness proof is different, because it must handle fair, rather than reliable, links—it can found in [15].

5.3 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 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 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.

protocols can be used here without modification, e.g. the leader election protocol in [2].

5.3.1 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* (*pc*). The *pc* serves two purposes. First, it allows the new leader to determine whether some value v may have been chosen, in which case the leader proposes v. We say that the correct leader will only propose a value that the progress certificate *vouches* for—we will discuss in Section 5.3.2 how a progress certificate vouches for a value. Second, the *pc* 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 process 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. Before examining progress certificates in more detail, let us examine how we would like to use them in the recovery protocol.

In the recovery protocol, the newly elected 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. Next, the leader follows the normal leader protocol to propose its value, and piggybacks 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).

5.3.2 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 voucher, λ can first sway f acceptors from the third group to "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.

To prevent progress certificates from being used twice as in the scenario described above, we make three changes. First, we only 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 to form a (value, number) pair. Acceptors now accept pairs rather than just values. Learners learn a pair o if they see that $\lceil (a+3f+1)/2 \rceil$ acceptors accepted it. We similarly alter the definition of *chosen* to apply to pairs, so (v, pn) is chosen if $\lceil (a+f+1)/2 \rceil$ correct acceptors have accepted it.

Third, we change the conditions under which acceptors accept a value (Figure 2). In addition to ignoring proposals with a proposal number lower than any they have seen (line 16), acceptors only accept one proposal for every proposal number (line 18) and they only change their accepted value if the progress certificate vouches for the new (value, number) pair (lines 20-21).

We are now ready to define progress certificates concretely. A progress certificate contains signed replies (v_i, pn) from $a-f$ acceptors (Figure 2, line 14). These replies contain that acceptor's currently accepted value and the proposal number of the leader who requested the progress certificate.

```
1   leader.onElected(newnumber):
2     number := max(number, newnumber)
3     if (not leader for this number) then return
4     send (QUERY,number,proof) to all acceptors
5     until get (REP, signed(value,number)) from a-f acceptors
6     PC := the union of these replies
7     if PC vouches for (v',number) then value := v'
8     onStart()
9
10  acceptors.onQuery(pn,proof): from leader
11    if (invalid proof or pn<largest_pn) then
12      return // ignore bad requests
13    largest_pn := pn
14    send (REP, signed(value,pn)) to the leader


15  acceptor.onPropose(value,number,progcert): from leader
16    if number!=largest_pn then
17      return // only listen to current leader
18    if accepted (v,pn) and pn=number then
19      return // only once per prop. number
20    if accepted (v,pn) and v!=value and
21    progcert does not vouch for (value,number) then
22      return // only change with change voucher
23    accepted := (value,number) // accepting
24    send (ACCEPTED,accepted) to all learners
```

Figure 2. FaB recovery pseudocode

Definition We say that a progress certificate $((v_0, pn), \ldots, (v_{a-f}, pn))$ *vouches* for the pair (v, 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 appears at least $\lceil (a-f+1)/2 \rceil$ times in the progress certificate, then the progress certificate vouches for that pair only. If there is no such pair, then the progress certificate vouches for any pair with the right *pn*. As we prove in the next section, progress certificates guarantee that if some pair (v, pn) is chosen, 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 0. Because of the poisonous write, the progress certificate allows the leader to propose any new value. To have "A" chosen, λ performs two steps: first, λ sends a new proposal ("A", 1) to the acceptors in the first group; then *l* sends ("A", 1) together with the progress certificate 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 pair.

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 the acceptors in the first group the progress certificate for proposal number 0, they

will refuse it as too low (line 16 of the protocol).

5.4 Correctness

We now proceed to prove that, for execution that are eventually synchronous, FaB Paxos solves consensus. Recall that a (value,number) pair is chosen iff $\lceil (a+f+1)/2 \rceil$ correct acceptors accept it.

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

Proof. Correct acceptors only accept values that are proposed. If a value is chosen, then it is accepted by correct acceptors so it follows that it was proposed. □

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

Proof. We prove this theorem by way of two lemmas.

Lemma 3. *For every proposal number pn, at most one value is chosen.*

Proof. Correct acceptors only accept one value per proposal number. Since to be chosen a (value, *pn*) pair must be accepted by at least a majority of the acceptors, at most one value is chosen per proposal number. □

Lemma 4. *If some pair $o = (v, pn)$ is chosen, then every progress certificate for proposal number $pn' > pn$ will vouch for o and no other value.*

Proof. Assume $o = (v, pn)$ is chosen; then, at least $c = \lceil (a+f+1)/2 \rceil$ correct acceptors Q have accepted o. Let pc be a progress certificate for proposal number $pn' > pn$. Consider the pairs contained in pc. Since these pairs are signed, they cannot have been manufactured by the leader; further, since by the protocol no correct acceptor would accept o if it had received pc with $pn' > pn$, all correct acceptors that accepted o must have done so before receiving pc. Since, by definition, pc contains $a-f$ pairs signed by acceptors, the number of these pairs that come from the acceptors in Q is at least $a-f+c-a$, which simplifies to $c-f = \lceil (a+f+1)/2 - f \rceil = \lceil (a-f+1)/2 \rceil$. By definition, then, pc vouches for o and no other value. □

In short, the lemmas state that each leader can choose at most one value, and if some leader chose a value then no subsequent leader can choose a different value. It follows directly that at most one value can be chosen. □

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

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

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

We say that a value is *stable* if it is learned by $\lceil (l+f+1)/2 \rceil$ learners. FaB Paxos only guarantees liveness when the execution is synchronous and there exists a time after which leaders that do not create a stable value are correctly deemed to be faulty. In this case, the leader election protocol ensures that a new leader is elected, and, further, that if the leader is correct then it will not be suspected. We assume this much in the the following lemma and proofs for CL1 and CL2.

Lemma 5. *Some value is eventually stable.*

Proof. Since the number of proposers p is larger than f, eventually either some value is stable or a correct leader α is elected. We show that if α is correct then some value will be stable. Let us, for a moment, assume reliable links.

The correct leader will gather a progress certificate 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.

However, links are not reliable, but only fair. The end-to-end retransmission protocol ensures that α will continue to resend its proposed value until it hears from $\lceil (l+f+1)/2 \rceil$ learners that they have learned a value—that is, until the value is stable. □

CL1. *Some proposed value is eventually chosen.*

Proof. By Lemma 5 eventually some value is stable, i.e. $\lceil (l+f+1)/2 \rceil > f$ learners have learned it. One of these learners is correct, and by CS3 a correct learner only learns a value after it is chosen. Therefore, the stable value is chosen. □

To prove CL1 it suffices to show 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 retransmissions will eventually stop.

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

Proof. By Lemma 5, some value v is eventually stable, i.e. $\lceil (l+f+1) \rceil \geq 2f+1$ learners eventually claim to have learned the value. Since at most f learners are faulty, at least $f+1$ of the learners that claim to have learned v are correct.

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 and receive $f+1$ matching responses. So eventually, all correct learners learn the chosen value. □

6 Parameterized FaB

Previous Byzantine Paxos protocols requires $3f+1$ processes and may complete in three communication steps when there is no failure; FaB requires $5f+1$ processes and may complete in two communication steps despite up to f failures—FaB uses the additional replication for speed. It may not be necessary to ensure 2-step operation even when all f processes fail; in some circumstances we may only be concerned with 2-step operation as long as, say, at most one process is faulty.

The Parameterized FaB protocol generalizes FaB by decoupling replication for fault tolerance from replication for speed. Parameterized FaB requires $3f+2t+1$ processes to guarantee safety and liveness despite up to f Byzantine failures, and guarantees common-case 2-step operation despite up to t Byzantine failures. In one extreme ($a = 5f+1$) the protocol has the same properties as non-parameterized FaB. In the other, the protocol only requires the minimal number of processes for consensus ($3f+1$) while at the same time allowing 2-step operation when there are no faults. For example, if seven machines are available, an administrator can choose between tolerating two Byzantine failures but slowing down at the first failure ($f = 2, t = 0$) or tolerating only one Byzantine failure but maintaining 2-step operation despite the failure ($f = 1, t = 1$).

Parameterizing FaB adds one additional round, which is only used when there are not enough correct acceptors to ensure 2-step progress. This round is similar to that in ordinary three-step Byzantine consensus protocols [2].

When an acceptor i accepts pair (v,r), it signs a message (v,r,i) and sends it to all acceptors. When an acceptor has proof that $q = \lceil (a+f+1)/2 \rceil$ acceptors have accepted the same value pair vp, it can generate a *commit proof* for vp. The commit proof is the set of q signed (v,r,i) messages from different acceptors but with the same value for v and r. The acceptors report the commit proof along with their accepted value to the learners.

A value pair vp is *chosen* if $\lceil (a+f+1)/2 \rceil$ correct acceptors have accepted vp or if $\lceil (a+f+1)/2 \rceil$ acceptors have a commit proof for vp. Learners learn vp when they know vp has been chosen.

We also modify the structure of progress certificates. When the new leader queries $n-f$ acceptors, the acceptors add to the signed pair they return either their commit proof, or a signed statement that they have none. These commit proofs (or non-commit proofs) are integrated into the progress certificate. A progress certificate now vouches for value v' if there is no value $d \neq v'$ contained $\lceil (a+f+1)/2-f \rceil$ times in the progress certificate and the progress certificate does not contain a commit proof for any value $d \neq v'$.

These changes 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.

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 a traditional three-round Byzantine consensus protocols.

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 accomodate existing leader election protocols (e.g. [2]).

The number of rounds of communication can be reduced down to three using *tentative execution* [2], an optimization proposed by Castro and Liskov for their PBFT protocol that applies equally well to FaB Paxos. As shown in Figure 3, 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 put. In case of conflict, tentative executions are rolled back and the requests are eventually re-executed in

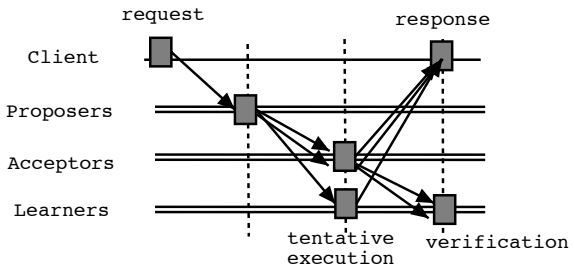

Figure 3. FaB state machine with tentative execution.

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 replicated state machine protocol can be further optimized to limit the amount of work in recovery and to require only $2f+1$ learners (reducing the development cost since each learner must have a different version of the program being replicated). We discuss these optimizations in our extended technical report [15].

8 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 2-step operation in the common case. Additionally, FaB Paxos in the common case does not require expensive digital signatures.

The price for common-case 2-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 [20], the design point occupied by FaB Paxos becomes much more intriguing.

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

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 *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 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. Best-case complexity of asynchronous byzantine consensus. Technical Report EPFL/IC/200499, EPFL, Feb. 2005.

[5] M. Fischer and N. Lynch. A lower bound for the time to assure interactice consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

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

[7] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, September 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, 2001.

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

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

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

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

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

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

[15] J.-P. Martin and L. Alvisi. Fast Byzantine Paxos. Technical Report TR-04-07, University of Texas at Austin, Department of Computer Sciences, February 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] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, Apr. 1997.

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

[20] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, pages 253–267. ACM Press, 2003.