2012 32nd IEEE International Conference on Distributed Computing Systems

When You Don't Trust Clients: Byzantine Proposer Fast Paxos

Hein Meling*, Keith Marzullo†, and Alessandro Mei‡
*Department of Electrical Engineering and Computer Science, University of Stavanger, Norway
†Department of Computer Science and Engineering, University of California San Diego, USA
‡Department of Computer Science, Sapienza University of Rome, Italy
Email: hein.meling@uis.no, marzullo@cs.ucsd.edu, mei@di.uniroma1.it

***Abstract*—We derive a consensus protocol for a hybrid failure model. In this model, clients are Byzantine faulty and servers are crash faulty. We argue that this model is well suited to environments where the servers run within one administrative domain, and the clients run outside of this domain. Our consensus protocol, which is derived from crash Paxos, provides low latency for client requests, tolerates any number of (Byzantine) faulty clients, up to 1/3 (crash) faulty servers, and does not rely on computing costly signatures in the common case. It can be used to build state machine replication that provides a highly available service.**

I. Introduction

State machine replication is a general approach for constructing fault-tolerant services, and a key protocol underlying state machine replication is consensus. The set of Byzantine failures is so large that it has been applied for masking the effects of compromised systems, and so Byzantine-tolerant consensus has been used to construct systems that are meant to ameliorate the effect of compromise (see [4] among many others). In the Byzantine model, there is no trust among processes: any process can behave in an arbitrarily faulty manner. However, in multi-site systems, processes executing within an administrative domain typically have a measure of mutual trust. This is because such processes share fate: for example, if a process in a domain is compromised, then other processes—perhaps all of them—can be compromised as well, and the local services they rely upon may be compromised. Byzantine-tolerant consensus can only mask the effects of a fraction of the processes exhibiting Byzantine failures. So, one has to be careful when thinking about using Byzantine-tolerant consensus to mask failures arising from processes that are in the same domain and have been compromised.

Insider attacks are examples of attacks that are hard to mask using Byzantine-tolerant consensus. In practice, other tools are used to detect and recover from an attack on an administrative domain. These tools, which include intrusion detection and sandboxing with virtual machines, are far from perfect, but they are regularly used in practical systems.

Outsider attacks are waged by communication. In this paper, we consider a service architecture that consists of three levels: (1) clients, which are untrusted; (2) proxies, which are processes that communicate directly with clients and so are vulnerable to outsider attack; (3) servers, that communicate with proxies and use replication to mask benign failures. The servers assume the proxies to be Byzantine faulty, because they can be compromised. But, the communication path between clients and proxies is narrow, reducing the ability of the proxies to be used to wage an attack on the servers. If such an attack on servers does happen, then it would be treated like an insider attack on the servers: using the tools mentioned above for detection and recovery.

We develop a consensus protocol, amenable for state machine replication, based on this service architecture. We call this protocol *Byzantine Proposer (BP) Fast Paxos* and develop it from Paxos with a set of refinements. BP Fast Paxos provides low latency for client requests, can tolerate any number of (Byzantine) faulty proxies, up to 1/3 (crash) faulty servers, and can protect itself against denial of service attacks waged by clients or proxies.

II. System Model

We assume the classic asynchronous system model in which the relative speed of processes and communication delay is unbounded, and clocks are not synchronized. The network is unreliable, allowing messages to be dropped, reordered or duplicated. However, we assume links are fair, meaning that if a message is sent infinitely often it will be received infinitely often. Our algorithms rely on reliable links, which can be implemented over fair links. A message receiver knows who the sender is; that is, messages must be authenticated in order to validate their origin. Note that, message authentication can be implemented efficiently by using a shared secret (symmetric) key for each node pair, instead of costly digital signatures.

The protocol is described, akin to Paxos [13], in terms of the following agent roles: proposers, acceptors, and learners. Paxos makes no assumption about the placement of these agents, but in a typical replicated state machine configuration, all server processes play all roles, while client processes interact with the proposers. In BP Fast Paxos, the choice of where to place the agents is dictated by our service architecture. Figure 1 illustrates our service architecture with an example agent configuration and their respective trust domains. In our architecture, a *proposer is a proxy* and is placed at the data center edge where it is more prone

1063-6927/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDCS.2012.38

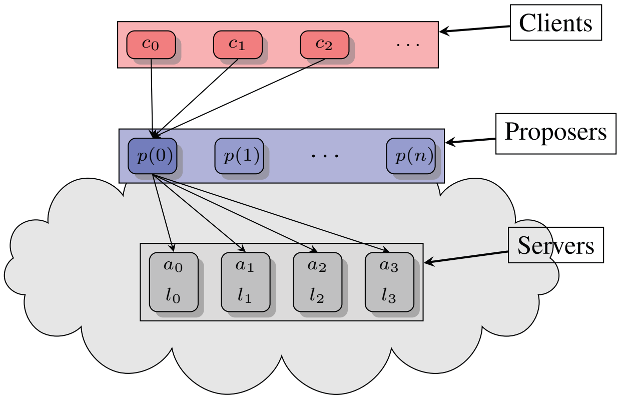

Figure 1. Proposers (proxies), $p(k)$, mapped to data center edge nodes. Rectangular boxes represent different trust domains. Rounded boxes represent Paxos agents, clients, or servers (acceptors a_i, learners l_i).

to attacks, and thus can become Byzantine. Acceptors are servers and may only experience crash failures because they are better protected, e.g. by firewalls. Servers are also learners. An alternative architecture in which proposers are placed at clients may also be possible, but is not considered in this paper.

There are issues that arise with such a service architecture in which clients and proposers are separated. For example, a proposer can construct a man-in-the-middle attack [22]; this can be thwarted by establishing a session key between the client and the servers using public-key based signatures (this is both secure and efficient, since the session key can then be used by the client to submit a long sequence of commands). A client needs to communicate with the proposer that has the role of leader (described below); this can be done, for example, by using multicast to all the proposers, by receiving a hint from a server, or by redirection among proxies. Each approach has its benefits and drawbacks, and are outside of the scope of this paper.

We assume that there are $n_p > t_p$ proposers, $n_a > 3t_a$ acceptors, and n_l learners, where t_p and t_a denote the maximum number of *Byzantine faulty proposers* and *crash faulty acceptors*, respectively. The number of learners n_l is application dependent, but our protocol requires that acceptors also be learners to detect misbehaving proposers. Hence, we have that $n_l \geq n_a$.

The protocol is a general consensus protocol, and it can be used to implement state machine replication. We assume the existence of mechanisms for agent membership management [16], for example as commands to the state machine. We also assume key distribution mechanisms to support message authentication and signatures. We assume that cryptographic primitives cannot be trivially circumvented. For simplicity, our initial description of BP Fast Paxos rely on digital signatures, but we later show that these can be replaced by MAC-based authenticators instead. Finally, we assume that Byzantine participants are not able to forge signatures or authenticators, and correct participants do not divulge any key information that would otherwise enable an attack.

As mentioned above, our protocol can tolerate an arbitrary number of faulty proposers. And even if all proposers are faulty, non-faulty learners will always observe a consistent system state. However, liveness can be violated if all proposers are faulty. Faulty proposers are detected using a muteness augmented $\Diamond S$ failure detector, as we explain in Section V-C. Moreover, a faulty client may propose disruptive commands that will be learned and executed by the system, but consistency among correct learners is never violated under the given assumptions. Prohibiting Byzantine faulty clients is of course impossible, but its risks can be reduced by using access control mechanisms on clients [19]. Applications using the protocol might also exploit semantic knowledge about the proposal values to detect misbehaving clients.

BP Fast Paxos is a refinement from Paxos, thus in the remainder of the paper, we refer to Paxos agents rather than to clients, proxies, and servers.

III. Consensus and Paxos

This section defines consensus and gives a brief overview of Paxos and Fast Paxos. The objective of a distributed consensus algorithm is to have a single value chosen among those proposed. Typically, *consensus* is stated in terms *safety* and *liveness* properties [14], [18]:

CS1 Only a proposed value may be chosen.
CS2 Only a single value is 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.

Note that the definition permits multiple values to be proposed, e.g. by a faulty proposer. An algorithm satisfying the safety properties is considered safe in the sense that all participants that learn the chosen value remain consistent with each other.

The term *round* is defined as a set of semantically related messages that may or may not conclude the consensus protocol. We say that the protocol solves consensus in some round. Due to asynchrony and failures, a consensus protocol may need to run several rounds to solve consensus. In the protocols, we shall use the variables rnd and $vrnd$ to denote round numbers.

We use the term *consensus instance* to refer to one among several executions of consensus. Each instance has its own set of variables and they may operate concurrently and independently. When using consensus to order messages for state machine replication, a sequence of consensus instances define the correct ordering to be executed by the replicas.

A. *Classic Paxos*

Paxos [13], [14] is described in terms of three separate agent roles: *proposers* that can propose values, *acceptors* that accept a value among those proposed, and *learners* that learn the chosen value. A process may take on multiple roles: in a typical configuration, all processes play all roles. Paxos is safe for any number of crash failures, and can make progress with up to t_a crash failures, given $2t_a + 1$ acceptors.

Every round is associated with a single proposer, which is the *leader* for that round. Proposers can start off rounds concurrently by sending a ⟨PREPARE⟩ message to acceptors, trying to make the value they propose chosen by the acceptors. Every round runs in two phases: (1) A proposer collects a majority of ⟨PROMISE⟩ messages in response to a previously sent ⟨PREPARE⟩ message; and (2) the proposer then sends ⟨ACCEPT⟩ messages for some value v to acceptors, who respond by sending ⟨LEARN⟩ messages to learners. The value v to select is the value with highest round among those provided in the ⟨PROMISE⟩ messages or if no votes are provided in the ⟨PROMISE⟩ messages, any value can be chosen. A complete classic Paxos algorithm is provided in our technical report [21].

In Paxos, acceptors are said to have *chosen* a value v, if a majority of acceptors have voted for v in the same round. Once a value has been chosen by acceptors in a round, no other value can be chosen in any other round. However, if there is no majority of acceptors that have voted for v, then the acceptors may vote for different values in other rounds. This will be an important issue when proposers may be Byzantine.

B. *Fast Paxos*

Fast Paxos [15] changed the communication pattern of Paxos by letting the proposer send its proposals directly to the acceptors, bypassing the leader and saving one message delay. In Fast Paxos, rounds are classified as *fast* or *classic*, depending on the round number.

The algorithm changes Paxos in several ways: (i) In fast rounds, the leader sends an ⟨ACCEPT-ANY⟩ message to acceptors. This enables acceptors to receive ⟨ACCEPT⟩ messages for any value from all proposers instead of just the leader. However, since proposers are not synchronized, they may propose differently to the acceptors, and thus we may have that a value is not chosen. This is called a *collision*. (ii) To make progress in this situation, there are two alternative recovery schemes. In *coordinated recovery*, acceptors send their ⟨LEARN⟩ messages to both learners and the leader. If the leader sees a collision, it starts another round by sending ⟨ACCEPT⟩ for a value based on those seen in ⟨LEARN⟩ messages. In *uncoordinated recovery*, acceptors instead send their ⟨LEARN⟩ messages to all other acceptors, allowing acceptors to immediately pick a value and send another ⟨LEARN⟩ based on the values from the previous round of ⟨LEARN⟩ messages without involving the leader. (iii) The rule the leader uses for picking the value v for ⟨ACCEPT⟩ messages must be modified to support multiple proposals. That is, if no value has enough votes, *any* value among those proposed is used. Acceptors in case of uncoordinated recovery, use the same rule to prepare a new ⟨LEARN⟩. However in this case, some additional restrictions can be applied to avoid further collisions [15]. (iv) Finally, in Fast Paxos the replication requirement is $n_a \geq 3t_a + 1$.

IV. DERIVING TRUST CHANGE PAXOS

As the first step in deriving BP Fast Paxos, we retain the crash failure assumption of Paxos, and modify it such that the trust relation between proposers and acceptors becomes more explicit. In Paxos, any proposer can start its round whenever it wishes. In *Trust Change (TC) Paxos*, proposers start non-0 rounds only when asked. A detailed TC Paxos algorithm is provided in our technical report [21].

A. *Moving the Responsibility of Initiating Rounds*

In TC Paxos, just like in Paxos, each round is assigned to a single proposer. The choice of the proposer for round i is determined by a deterministic mapping $p : B \rightarrow P$, where B is the set of round numbers and P is the set of proposers. In this paper we assume that B is the set of natural numbers. Mapping p can be arbitrary as long as it maps infinitely many rounds to every proposer in P. For simplicity, we can assume that proposers have assigned identities $0, 1, \ldots, |P| - 1$, where $|P| = n_p$. Then, we can choose mapping p such that $p(i) = i \pmod{|P|}$.

In TC Paxos, the initiative of starting off a round lies with the acceptors. The acceptors *change trust* from one proposer to the next if the current one is faulty. This is done with ⟨TRUSTCHANGE⟩ messages, which in turn, trigger ⟨PROMISE⟩ messages sent to the newly trusted proposer. As in Paxos, when a proposer has received a quorum of such ⟨PROMISE⟩ messages, it properly chooses a value and sends a corresponding ⟨ACCEPT⟩ message. Moreover, accepting and learning are also identical to Paxos. By itself TC Paxos is interesting as an alternative way of implementing Paxos. Moreover, it is an important lead-up to BP Fast Paxos, our version of Paxos that can tolerate Byzantine proposers.

The ⟨TRUSTCHANGE⟩ messages are the equivalent of the ⟨PREPARE⟩ messages of Paxos. Indeed, just like in Paxos, a ⟨TRUSTCHANGE⟩ message is followed by a ⟨PROMISE⟩ message from the acceptors to the proposers. The idea of the following simple lemma is to show that TC Paxos is safe by reducing TC Paxos to Paxos and relying on the fact that Paxos is safe.

Lemma 1: TC Paxos has Properties CS1, CS2, and CS3.

Proof: We show that, at the message level, every legal execution of TC Paxos can be transformed into a legal execution of Paxos. Since we know that Paxos is safe, i.e. it has Properties CS1, CS2, and CS3 [13], [14], we argue that TC Paxos must be safe as well.

Take a legal execution of TC Paxos at the message level. Every $\langle$TRUSTCHANGE$\rangle$ message that changes trust to round $i + 1$ and that is received by acceptor a is replaced by a $\langle$PREPARE$\rangle$ message from proposer $p(i + 1)$ to acceptor a with the same round. It is easy to see that the resulting execution is a legal execution of Paxos. Since Paxos has Properties CS1, CS2, and CS3, TC Paxos has these properties as well. ■

B. Progress in Trust Change Paxos

To achieve progress, Paxos implementations usually rely on a leader election protocol that is used to select a "distinguished" proposer that is the only one supposed to be executing rounds. Without a unique leader, progress is not guaranteed. More formally, Paxos relies on the failure detector Ω [5], which is the weakest failure detector to solve consensus and that, eventually, indicates the same correct leader to all the correct processes. By using Ω, eventually a value is chosen, provided that $n_p > t_p$ and $n_a \geq 2t_a + 1$.

In TC Paxos, it is more natural to use failure detector $\Diamond S$ (which is equivalent to Ω). Failure detector $\Diamond S$ has strong completeness (eventually every process that crashes is permanently suspected by every correct process) and eventual weak accuracy (eventually some correct process is not suspected by any correct process). For a more formal definition see [6]. The $\Diamond S$ abstraction is used by the acceptors to monitor the status of the proposers and, eventually, agree on the same correct proposer to trust.

Lemma 2: Using $\Diamond S$ and under the assumption that $n_p > t_p$ and $n_a \geq 2t_a + 1$, TC Paxos satisfies Properties CL1 and CL2.

Proof: With $\Diamond S$, eventually (say after time t) every process that crashes is permanently suspected by every correct process and there will be at least one correct proposer p that is not suspected by any correct acceptor. Let i be the highest round at which any acceptor is at time t, and let $i' \geq i$ be a round associated with proposer p. First, note that any acceptor that has made it to round i' will not change its trust beyond round i' because proposer p is not suspected by any acceptor using $\Diamond S$. Therefore, the number of trust changes in the execution is finite.

Let i'' be the highest round reached in the execution: we know such a round exists since the number of trust changes is finite. We also know that it is associated with a correct process, since $\Diamond S$ has strong completeness. When the first acceptor reaches round i'', it sends $\langle$TRUSTCHANGE$\rangle$ messages to all the other acceptors to change trust to round i''. Since we assume reliable links, eventually every acceptor will receive the message and change trust to round i''. Finally, since the proposer associated with round i'' is correct, it will propose a value that, since $n_a \geq 2t_a + 1$, will be chosen and learned by correct learners. ■

V. DERIVING BYZANTINE PROPOSER FAST PAXOS

We now relax the failure assumption of the proposers. Specifically, we now assume proposers may be Byzantine faulty, while the crash failure assumption is retained for acceptors and learners. This assumption precludes implementing acceptors and proposers in the same process. We derive BP Fast Paxos, by strengthening TC Paxos to mask Byzantine proposers. In TC Paxos, acceptors are responsible for detecting faulty proposers. We leverage this structure in BP Fast Paxos, by augmenting the responsibility of acceptors to also detect Byzantine proposers. Recovery from a faulty proposer is simply handled by reusing the trust change mechanism introduced in TC Paxos. BP Fast Paxos is shown in Algorithms 1 and 2.

Starting from TC Paxos, we now assume proposers may behave maliciously with the intention to violate safety. Acceptors in TC Paxos only process $\langle$ACCEPT$\rangle$ messages for round i from proposer $p(i)$. To ensure this also in BP Fast Paxos, we require channels between proposers and acceptors to be authenticated. Since acceptors only process $\langle$ACCEPT$\rangle$ messages for round i from proposer $p(i)$, there are only two ways in which the proposer can misbehave in terms of safety: Case (i) it can choose an arbitrary value in the $\langle$ACCEPT$\rangle$ message, and Case (ii) it can send different $\langle$ACCEPT$\rangle$ messages to different acceptors (*equivocation*).

Case (i) is problematic because the proposer can cause the chosen value to change arbitrarily from one round to another, and Case (ii) can lead to starvation or to violation of safety. Although in TC Paxos, an acceptor will only process $\langle$ACCEPT$\rangle$ messages from the current proposer, the same proposer can portray to be in different rounds. For example, consider proposer $p(0) = p(n_p)$. This proposer can send both $\langle \text{ACCEPT}, 0, v \rangle$ and $\langle \text{ACCEPT}, n_p, v' \rangle$, and thus change the chosen value of acceptors causing learners to see conflicting $\langle$LEARN$\rangle$ messages for two different values. This can violate safety if only a subset of learners see the learn for v, while another subset of learners see v'. Another situation where a faulty proposer may change the value is the following: Assume acceptors have chosen v in round 0, followed by suspecting $p(0)$ and thus sending a $\langle$PROMISE$\rangle$ to $p(1)$, a Byzantine proposer. At this point, $p(1)$ can malevolently change the value to v' in the $\langle \text{ACCEPT}, 1, 0, v' \rangle$ message. Learners may see more proposals for newer rounds, but newer rounds should have the same value v as was previously voted for, because some learners might have chosen v in an earlier round. Finally, since learners could potentially see different values for some round i, they may be unable to decide on the consensus value (starvation) since a majority is required.

A. Case (i): Introducing Signatures

Since the proposer cannot be trusted, acceptors must detect misbehavior and trigger a trust change to prevent a faulty proposer from violating safety, as explored in the examples

Algorithm 1 Proposer c

```
Initialization:
$A$                                                   {Set of acceptors}
$crnd \leftarrow 0$                                   {Current round number}

on event $crnd = 0$ and $c = p(0)$
  $cval \leftarrow \text{pickAny}()$                  {Propose a value}
  send $\langle \text{ACCEPT}, crnd, cval \rangle$ to $A$

on $\langle \text{PROMISE}, [rnd, vrnd, vval]_a \rangle$
  if $rnd > crnd$ then
    $crnd \leftarrow rnd, \quad MV \leftarrow \emptyset, \quad proof \leftarrow \emptyset$   {New round—this proposer is trusted}
  $MV \leftarrow MV \uplus (vrnd, vval)$              {Collect in multiset $MV$ last votes}
  $proof \leftarrow proof \cup [rnd, vrnd, vval]_a$   {Collect proof from acceptor $a$}
  if $|MV| \geq n_a - t_a$ then                       {Got enough promises from acceptors?}
    $QV \leftarrow \text{pickLargest}(MV)$            {Pick all votes with largest $vrnd$}
    if $\exists x \in QV : \text{count}_{QV}(x) \geq n_a - 2t_a$ then   {Does vote for $x$ occur at least $n_a - 2t_a$ times?}
      $cval \leftarrow x$                             {Pick the vote value in $QV$}
    else
      $cval \leftarrow \text{pickAny}(QV)$            {If $QV \neq \emptyset$, pick any vote value in $QV$; otherwise, propose one}
    send $\langle \text{ACCEPT}, crnd, cval, proof \rangle$ to $A$
```

Algorithm 2 Acceptor a

```
Initialization:
$P$, $A$, $L$                                         {Sets of proposers, acceptors, and learners}
$rnd \leftarrow 0$                                    {Current round number}
$vrnd \leftarrow \bot$                                {Last voted round number}
$vval \leftarrow \bot$                                {Value of last voted round}
$ML \leftarrow \emptyset$                             {Multi-set of $\langle \text{LEARN} \rangle$ messages}

on $\langle \text{TRUSTCHANGE}, n \rangle$ with $n > rnd$ from acceptor $a$   {Change trust to a new proposer}
  $rnd \leftarrow n$                                  {The next round number}
  send $\langle \text{PROMISE}, [rnd, vrnd, vval]_a \rangle$ to $p(rnd)$

on $\langle \text{ACCEPT}, n, v, proof \rangle$ with $n \geq rnd \wedge n \neq vrnd$ from proposer $c = p(n)$
  if $n = 0 \vee \text{verify}(n, v, proof)$ then     {If $n > 0$, check signatures and consistency of $n$, $v$, $proof$}
    $rnd \leftarrow n, \quad vrnd \leftarrow n, \quad vval \leftarrow v$
    send $\langle \text{LEARN}, n, v \rangle$ to $L$
  else
    send $\langle \text{TRUSTCHANGE}, n+1 \rangle$ to $A$   {Invalid proof; notify acceptors to change trust}

on $\langle \text{SUSPECT}, p(rnd) \rangle$ from $\Diamond S_a$   {$\Diamond S_a$ suspects the current proposer $p(rnd)$ has crashed}
  send $\langle \text{TRUSTCHANGE}, rnd+1 \rangle$ to $A$   {Change trust to another proposer}

on $\langle \text{LEARN}, n, v \rangle$ from acceptor $a$   {Algorithm executed by learners; acceptors are also learners}
  $ML \leftarrow ML \uplus (n, v)$                    {Store learn message for round $n$ and value $v$}
  if $\exists n, v, v' : (n, v) \in ML \wedge (n, v') \in ML$ then   {Byzantine proposer?}
    send $\langle \text{TRUSTCHANGE}, n+1 \rangle$ to $A$   {Change trust to another proposer}
  else
    if $\exists (n, x) \in ML : \text{count}_{ML}(x) \geq n_a - t_a$ then   {Quorum for value $x$ in $ML$?}
      send $\langle \text{DECIDED}, x \rangle$ to app   {We're done!}
```

above. In the first step, we address Case (i) above by introducing the following modifications to TC Paxos:

1) When requesting change of trust to a new proposer, an acceptor a signs the triplet, $\sigma_a = [rnd, vrnd, vval]_a$, that it voted for in the most recent round, $vrnd$. This signature is then sent as part of the trust change as:

$\langle$PROMISE, $[rnd, vrnd, vval]_a\rangle$.

2) On receiving enough $\langle$PROMISE$\rangle$ messages, a new proposer can send an $\langle$ACCEPT$\rangle$ to acceptors. However, it must now include a $proof = \{\forall a \in A : \sigma_a\}$ containing the signatures from acceptors A, attesting to the proposer's action to propose some value. Since up to t_a acceptors may fail, the proposer is only required to include $n_a - t_a$ signatures in the $proof$.
3) Acceptors must verify that the value v proposed in the $\langle$ACCEPT$\rangle$ from the new proposer is supported by votes from previous rounds, and *that may have been chosen*. The $proof$ is used to verify this. If a violation is detected, another trust change is triggered.

These changes are aimed at preventing a new proposer from changing the vote arbitrarily. The first two changes do not alter any significant behavior with respect to TC Paxos, they merely add an extra field to two messages that enable the check in Item 3 (represented by the verify() call in Line 11 of Algorithm 2.) However, since the proposer may be Byzantine faulty, the $proof$ constructed in Item 2 must be such that it can be checked by acceptors (Item 3). If the proposer tries to change the last value that has been voted for by the acceptors in previous rounds, a call to verify() will detect this. In this case, the acceptor notifies the other acceptors (Line 15) that a trust change is needed.

We now argue that the above changes to TC Paxos still preserve safety despite proposers trying to change the chosen value. First consider a correct proposer, in which case the acceptors receive an $\langle$ACCEPT$\rangle$ with a correct $proof$, and therefore Lines 12-13 are executed. This is identical to a TC Paxos execution. TC Paxos also handles crash faulty proposers. Next consider a semi-Byzantine proposer that does not equivocate.

Lemma 3: A non-equivocating proposer cannot undetectably change a chosen value.

Proof: By contradiction of two cases. Suppose there is a proposer $p(k)$ that can undetectably change a chosen value v to v'.

First, let $p(0)$ be the initial proposer of value v, and for convenience let $k = 1$. Let $p(0)$ be faulty, causing a majority of acceptors that have voted for v, to issue a trust change by sending $\langle$PROMISE, $[1, 0, v]_a\rangle$ to $p(1)$, vouching for v in round 0. Assume $p(1)$ is faulty and changes the value to v' by sending $\langle$ACCEPT, $1, v', proof\rangle$. However, since a majority of acceptors vouch for v, there must be at least one entry in $proof$ that vouches for $(0, v)$, and thus the acceptors can detect the violation. This contradicts the initial assumption that $p(1)$ can change a chosen value. By the same argument, it is easy to see that this also holds for any $k > 1$ too.

Second, let $k = 0$ and suppose $p(0) = p(n_p)$ is faulty and that it can change the chosen value from $(0, v)$ to (n_p, v'). That is, the same proposer is proposing for both round 0 and n_p, because proposers are reused after n_p trust changes. Assume $p(0)$ tries to change the value, after its initial value v was chosen by acceptors, by sending $\langle$ACCEPT, $n_p, v', proof\rangle$. However, since only acceptors can provide the signatures necessary to construct the $proof$, this prevents the proposer from having v' chosen by acceptors. Moreover, if an acceptor voted for v in a previous round, e.g. $n_p - 1$ or 0, it would only sign a promise such that the $vval = v$, hence preventing a proposer from collecting a $proof$ to support v'. Thus acceptors detect a violation, contradicting the initial assumption. ■

B. Case (ii): Detecting Equivocation

Our changes so far apply to rounds $i > 0$, i.e. when a trust change was necessary. For round 0, acceptors simply send $\langle$LEARN$\rangle$ messages based on what they received in the $\langle$ACCEPT$\rangle$ message; there is no $proof$ that can be checked at this stage since acceptors have only seen one $\langle$ACCEPT$\rangle$.

This brings us to Case (ii) where a faulty proposer may send $\langle$ACCEPT$\rangle$ for different values to different acceptors. This is essentially the same problem that occurs in Fast Paxos [15], when in fast rounds different acceptors can vote to accept different values in the same round, possibly causing no value to be chosen. This can happen if acceptors receive proposals from different proposers in different orders, and is called a collision. Fast Paxos offers two ways to recover from this problem, as mentioned in Section III-B. Thus, the next step of our derivation is to make modifications similar to Fast Paxos to tackle this problem of acceptors seeing different values:

1) Add equivocation detection and recovery.
2) Change the (proposer) leader's rule for picking a value v.
3) Increase the number of acceptors to $n_a \geq 3t_a + 1$.

Clearly, since the current proposer (leader) may send different values to different acceptors, we need some way of detecting and recovering from this misbehavior. In Fast Paxos, a coordinated recovery could be used, in which the leader is responsible for detection. This obviously does not work when the leader might be Byzantine; it would be responsible for detecting its own misbehavior. Therefore, to recover, a combination of coordinated and uncoordinated recovery is used, where acceptors detect misbehavior and the new proposer coordinates recovery; it works as follows: Let *acceptors be learners*, i.e. acceptors send each other $\langle$LEARN$\rangle$ messages, in order to detect equivocation in round i; see Lines 18-24 of Algorithm 2. If equivocation is detected (Line 20), then that is proof that $p(i)$ is faulty, and a trust change to round $i+1$ is appropriate. This last part is different from uncoordinated recovery, where acceptors would try to choose a value for the next round without using the leader. However in BP Fast Paxos, observing different votes from the same proposer simply means that it must be faulty and should be replaced with a new proposer. The new proposer will coordinate recovery by proposing again.

This change works in concert with our first changes above to strengthen the trust change with signatures, allowing acceptors to provide evidence of misbehavior to a new proposer during a trust change. Moreover, similarly to Fast Paxos, we must change the rules for picking a value v for the next round when there is a non-majority. Our rule is the same as in Fast Paxos; see Lines 14-17 of Algorithm 1. As in Fast Paxos, Item 3 above is also necessary because the proposer must be able to choose a unique value that *may have been or will be chosen*. To understand the necessity for this change, consider the following example in which a faulty proposer, proposes values $\{v, v, v'\}$ to the acceptors followed by $\langle$LEARN$\rangle$ messages to learners. Now a set of learners could see the two $\langle$LEARN$\rangle$ messages for value v and decide, while acceptors on seeing this misbehavior starts a trust change. However, if the new proposer only sees two $\langle$PROMISE$\rangle$ messages, one for each of v and v', it must pick one of the two. If it picks v' in the $\langle$ACCEPT$\rangle$ message, followed by corresponding $\langle$LEARN$\rangle$ messages, then another set of learners that did not learn v, e.g. due to message loss, may now decide v' instead, violating safety.

We now provide a correctness proof for the protocol.

Theorem 1: BP Fast Paxos satisfies the consensus safety properties CS1, CS2, and CS3.

Proof: Property CS1 is easy to check by examining Algorithms 1 and 2, used by the proposers and acceptors, respectively. Property CS3, like in TC Paxos, follows directly from the assumption of fair-loss links. Hence, we focus on Property CS2. We prove, by induction, the following *safety property* of BP Fast Paxos: If any acceptor *votes* for a value v in round i, then no value $v' \neq v$ can be *chosen* in rounds $0, \ldots, i-1$. It is easy to see that this safety property implies Property CS2.

It is trivial to see that the safety property holds for round $i = 0$ (the *base case*). Consider $i > 0$ and assume that the safety property holds for every round before i (the *inductive hypothesis*). The goal is to prove that it holds for i as well. Assume that acceptor $a \in A$ has voted for value v in round i. Since acceptors are not Byzantine, acceptor a must have received an $\langle$ACCEPT$\rangle$ message for round i carrying the *proof* of promises from a subset $A' \subseteq A$ of acceptors, where $|A'| \geq n_a - t_a$. The proofs are checked, so the senders in A' are authenticated acceptors and the signed triplets $[rnd, vrnd, vval]_{a'}$, $a' \in A'$, are not forged. By Lemma 3, this is sufficient to prevent a faulty proposer from changing the chosen value.

Let j be the highest $vrnd$ in the triplets. Clearly, $j < i$. First, no value can be chosen in rounds $j+1, \ldots, i-1$, if $j \neq \perp$, and in rounds $0, \ldots, i-1$, if $j = \perp$. Indeed, we know that the acceptors in A' promise not to vote in rounds before i, that $|A'| > t_a$ since $n_a \geq 3t_a + 1$, and so the remaining $|A \setminus A'| < n_a - t_a$ are *not enough* to form a quorum in rounds $< i$. In particular, if $j = \perp$, then we are done with the proof of the inductive step. So, let's assume $j \geq 0$ and proceed.

Let Q be the multiset of votes in round j provided by the acceptors in A'. Since $j \geq 0$, we know that $Q \neq \emptyset$. Acceptor a has voted for value v in round i, therefore, either (i) v appears at least $n_a - 2t_a$ times in Q, or (ii) $v \in Q$ and no value appears at least $n_a - 2t_a$ times in Q (see Lines 14-17 of Algorithm 1). In case (i) v is the *only* value that can still be chosen in round j, indeed the acceptors that have *not* voted v are at most $2t_a$, and since $n_a \geq 3t_a + 1$, that is *not enough* to form a quorum of $n_a - t_a$ acceptors on a value $v' \neq v$ in round j. In case (ii) no value can be chosen in round j, indeed no quorum of $n_a - t_a$ acceptors is possible on any value since $|A \setminus A'| \leq t_a$. In other words, no value $v' \neq v$ can be chosen in round j.

Finally, by the inductive hypothesis, we also know that no value $v' \neq v$ can be *chosen* in rounds $0, \ldots, j-1$, since $j < i$ and value v has been voted by at least one acceptor in round j. Hence, no set of proposers can cause acceptors to choose differently by equivocating. This concludes the inductive step and the proof. ■

C. Liveness and Resilience to Performance Attacks

Next we provide an informal argument that our protocol is live. From a progress perspective, BP Fast Paxos operates in the same way as TC Paxos, except that proposers can be Byzantine faulty. In general, implementing failure detection with Byzantine faulty participants is not trivial. However, since the acceptors implement the failure detector over the proposers, and acceptors are only crash faulty, we only need to ensure that the $\Diamond S$ failure detector also suspects mute Byzantine faulty processes as crashed processes. Acceptors detect proposers sending incorrect information—and send $\langle$TRUSTCHANGE$\rangle$ messages to the other acceptors (Line 17) to replace a faulty or slow proposer. Moreover, if Byzantine behavior (e.g. equivocation) is detected by an acceptor, then it also sends a $\langle$TRUSTCHANGE$\rangle$ message to the other acceptors. Thus, since TC Paxos is live under $\Diamond S$ (Lemma 2), BP Fast Paxos is live under a $\Diamond S$ failure detector that detects mute proposers.

As alluded to by the above argument, our protocol can also support a mechanism to cope with *performance attacks* [3], [9] and *denial of service attacks* by Byzantine proposers. Indeed, if a Byzantine proposer delays message sending, or acts in such a way so as to reduce performance or otherwise prevents the system from making progress, this can be detected by the acceptors. For instance, if a proposer is not forwarding the expected number of commands (for whatever reason), it can easily be replaced by another proposer through a trust change. The change is once again initiated by an acceptor, and the reason for the change does not require any rigorous evaluation of misbehavior. It can simply be viewed as a rotation of the leader, as was done in [2], [9]. Developing this idea further is outside of the scope of this paper.

D. Optimizations

In BP Fast Paxos we can replace costly digital signatures [4] with message authentication codes (MACs). This is feasible since digital signatures provide the stronger non-repudiation property that is not needed in our protocol.

Acceptors are both signers and verifiers of $\langle$PROMISE$\rangle$ messages exchanged during a trust change. To sign a $\langle$PROMISE$\rangle$, an acceptor simply generates a MAC which includes the $\langle$PROMISE$\rangle$ and its own identifier. A correct proposer will collect the signed promises from acceptors into a *proof* vector. Once provided with a *proof* from the new proposer, acceptors can verify the correctness of its own vector component in the *proof*. By assumption, acceptors only fail by crashing and will not disclose the authenticator key. Thus, since there is trust among the acceptors, it suffices to use a single shared authenticator key among them.

We note that BP Fast Paxos does not rely on *proof* vectors for the common case. (Of course we still need a single MAC to authenticate the channel.) To ensure common case operation over multiple consensus instances in a state machine implementation, it is necessary to reconfigure the proposer function $p(i)$. This is since common case operation is restricted to $p(0)$. This reconfiguration can be implemented similarly to a Paxos membership reconfiguration [16].

VI. DETECTING BYZANTINE ACCEPTORS

Thus far we have assumed that acceptors can only fail by crashing, a reasonable assumption if acceptors can be adequately protected from becoming compromised, e.g. with firewalls. In this section, we first analyze how a Byzantine acceptor can compromise the safety of BP Fast Paxos, followed by another change that enables correct acceptors to *detect* a limited number of Byzantine acceptors. In this way, the protocol can have a role in intrusion detection. Indeed, if services are implemented on replicated servers by using BP Fast Paxos with detection of Byzantine acceptors, the detection of a misbehaving acceptor can be useful for an intrusion detection system, since this event can indicate that the system has been compromised.

We first show that BP Fast Paxos cannot mask even a single Byzantine acceptor, despite its high redundancy requirement on acceptors ($3t_a + 1$). This is because a Byzantine acceptor is indistinguishable from a Byzantine proposer in round 0, making it impossible to determine which of the two agents to remove.

A Byzantine acceptor can simply send an arbitrary value $v' \neq v$ in its $\langle$LEARN$\rangle$ message to the learners (i.e. acceptors), where v is the value sent by (correct) proposer $p(0)$ in its $\langle$ACCEPT$\rangle$ message. By this action, a Byzantine acceptor can force a trust change, since this misbehavior is detected by the (correct) acceptors, whom has no other choice but to trigger a $\langle$TRUSTCHANGE$\rangle$ (Line 21 of Algorithm 2), blaming the misbehavior on $p(0)$. Eventually $p(1)$ takes over, and propose a value supported by round 0. This is problematic because some learners may decide the correct value v, while others may never decide because the faulty acceptor is not removed.

However, assuming at most t_a faulty acceptors, these cannot cause the learners to decide on inconsistent values since a learner will only decide if it can collect $n_a - t_a$ identical values. Next however, we consider a more severe example in which a faulty acceptor a_0 collude with a faulty proposer $p(0)$ in order to change the chosen value. There are four acceptors. Assume $p(0)$ proposes the values v' to $\{a_0, a_1\}$ and v to $\{a_2, a_3\}$. Since a_0 is faulty, it can send a $\langle$LEARN$\rangle$ for v instead of v'. The other acceptors are correct, and send their appropriate $\langle$LEARN$\rangle$ messages. Let Q be a subset of learners that see a majority of $\langle$LEARN$\rangle$ messages for v: these learners can decide v in round 0. (We may assume $\langle$LEARN$\rangle$ messages for v' is delayed or lost and not yet retransmitted.) Now, assume that acceptors move their trust to $p(1)$. This can happen for any reason, given the unreliable failure detector. In response to this, each acceptor send a $\langle$PROMISE$\rangle$ message and sign their votes as follows: $\{\sigma_{a_0}(v'), \sigma_{a_1}(v'), \sigma_{a_2}(v), \sigma_{a_3}(v)\}$. Note that faulty acceptor a_0 has changed its vote. The new proposer must now either pick v or v' non-deterministically, and so $p(1)$ could send $\langle$ACCEPT$\rangle$ for v'. Let R be a subset of learners distinct from Q. Learners in R may now see $\langle$LEARN$\rangle$ messages for v' and decide on v'. This violates CS2.

It isn't surprising that such a scenario exists: a lower bound on two-step Byzantine consensus is one that needs at least $5t_a + 1$ acceptors [18]. BP Fast Paxos is two-step, but use only $3t_a + 1$ acceptors. To remain two-step and mask Byzantine acceptors, changing BP Fast Paxos would require $5t_a + 1$ acceptors, essentially making it equivalent to FaB [18]. The other alternative is to increase the number of steps along with signatures, taking us in the direction of PBFT [4].

The scenario above exists because our protocol cannot distinguish between a faulty proposer and faulty acceptor(s). To solve this problem, we need to identify the source of misbehavior. Thus in the following, we sketch a simple change that enables identifying the misbehaving Paxos agent(s).

Hence, we require that proposer $p(0)$ include a signature, $\sigma_{p(0)}(0, v)$ in its round 0: $\langle \text{ACCEPT}, 0, v, \sigma_{p(0)}(0, v)\rangle$ message. Acceptors forward the signature in its $\langle \text{LEARN}, 0, v, \sigma_{p(0)}(0, v)\rangle$ message. This will allow learners/acceptors to verify that the value v originated from $p(0)$, and prevents faulty acceptors from undetectably changing the value. Moreover, if the proposer is faulty by equivocation (on $\langle$ACCEPT$\rangle$ messages; detected by observing different $\langle$LEARN$\rangle$ messages), the acceptors can collect a proof of misbehavior [2] from the discrepancies between pairs of signed $\langle$ACCEPT$\rangle$ messages. The approach enables correct acceptors to identify faulty acceptors and/or a faulty proposer, and either change trust or replace faulty acceptors. Exploring the latter is beyond the scope of this paper.

Note that, although this extra signature[1] in the common case execution path does introduce additional latency due to the cost of preparing a signature at $p(0)$, we do not have to verify the signature at acceptors unless equivocation in the $\langle$LEARN$\rangle$ messages is detected. Thus, the cost of signature verification on the server-side is avoided in the common case. Moreover, it is also possible to leverage matrix signatures [1] to provide the non-repudiation property and thus reduce the cost of preparing a signature at $p(0)$. Also, since we already use an all-to-all communication pattern for sending $\langle$LEARN$\rangle$ messages between acceptors, this would not add additional latency. A similar approach is taken by Clement et al in [8].

VII. Related Work

Since PBFT [4] revived the interest in Byzantine fault tolerance by leveraging message authenticators to improve performance, numerous works have explored a range of other optimizations. These include reduced redundancy overhead, further performance improvements, and retaining good performance even during attacks (resilience to attacks).

A number of techniques have been proposed to reduce the steep redundancy requirement of BFT [24], [7], [23], [10]. One approach of particular interest, for use with our protocol, is to separate the agreement and execution phase of the BFT state machine [24]. For instance, while BFT typically requires $3t_a + 1$ replicas in the agreement phase, only $2t_a + 1$ replicas need to execute state machine commands. Hence, the cost in terms of execution replicas can be reduced significantly. BP Fast Paxos is an agreement protocol, and can also take advantage of such separation to reduce the replication cost of execution.

In terms of latency, BP Fast Paxos requires two communication steps to complete consensus in the common case. FaB [18] offers similar latency, and is shown to be optimal when also acceptors can be Byzantine. However, in the case of FaB, $5t_a + 1$ replicas are necessary for agreement, and $3t_a+1$ for execution. Zyzzyva [12] on the other hand, offers speculative execution of state machine commands in the common case by letting the client accept a reply once all $3t_a+1$ replies have been received. For this common case behavior, BP Fast Paxos would need one extra communication step compared to Zyzzyva with similar cryptographic costs. However, this comes at additional complexity and cost when speculation fails (e.g. not all replies are received) and in the view change procedure. Also, Zyzzyva is optimized for BFT state machine replication only, while BP Fast Paxos is a general consensus protocol applicable to other application domains.

Aardvark [9] and Prime [3] are both leader-based state machine protocols that focus on retaining performance during attacks (or failures in general). BP Fast Paxos is also leader-based, and can easily replace the leader if its performance is unsatisfactory.

BP Fast Paxos [20], [21] extends Paxos [13] and Fast Paxos [15] by separating the failure model assumed for the different Paxos agents. The proposers can behave arbitrarily and acceptors are assumed to be only crash faulty. To our knowledge, assuming different failure models for the different Paxos agents has not been explored previously. However, in RAM [17] servers in different administrative domains were assumed mutually suspicious of each other. Aiyer et al proposed the BAR model [2] for cooperative services, in which peers could take on three different types of behavior: Byzantine, Altruistic, or Rational (selfish). BAR is a more general model than ours and requires more complex protocols, typically involving incentive-based protocols. Guerraoui et al [11] propose a modular framework for developing BFT protocols, which could be used to develop our protocol.

VIII. Conclusions

This paper presents BP Fast Paxos, a consensus protocol for a service architecture designed for clients that can wage attacks. Clients communicate through proxies that can be Byzantine faulty, while the servers that run in their own domain are crash faulty. Compromised servers are dealt with like insider attacks, using the detection and recovery tools that are now used in practical systems. BP Fast Paxos is two-step and safe even when all proposers (proxies) are Byzantine faulty, and does not require costly digital signatures among the proposers, acceptors and learners for the common case.

References

[1] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement. Matrix signatures: From macs to digital signatures in distributed systems. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 16–31, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. Bar fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 45–58, New York, NY, USA, 2005. ACM.

[3] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Secur. Comput.*, 8:564–577, July 2011.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43:685–722, July 1996.

[1] The signature scheme used here must have non-repudiation properties, since a faulty acceptor can orchestrate a man-in-the-middle attack preventing correct acceptors from identifying the source of misbehavior.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 189–204, New York, NY, USA, 2007. ACM.

[8] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[9] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[10] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.

[11] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The Next 700 BFT Protocols. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 363–376, New York, NY, USA, 2010. ACM.

[12] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27:7:1–7:39, January 2010.

[13] L. Lamport. The part-time parliment. *ACM Trans. on Comp. Syst.*, 16(2):133–169, 1998.

[14] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[15] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[16] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41:63–73, March 2010.

[17] Y. Mao, F. Junqueira, and K. Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *Fifth Workshop on Hot Topics in Dependable Systems (HotDep'09)*, Estoril, Lisbon, Portugal, June 2009.

[18] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3(3):202–215, July 2006.

[19] J.-P. E. Martin. *Byzantine Fault-Tolerance and Beyond*. PhD thesis, University of Texas, Austin, December 2006.

[20] K. Marzullo, H. Meling, and A. Mei. Brief Announcement: When You Don't Trust Clients: Byzantine Proposer Fast Paxos. In *Proceedings of the 25nd International Symposium on Distributed Computing*, volume 6950 of *DISC '11*, pages 143–144, Rome, Italy, September 2011. Springer.

[21] H. Meling, K. Marzullo, and A. Mei. When You Don't Trust Clients: Byzantine Proposer Fast Paxos. Technical report, University of California, March 2012.

[22] B. Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[23] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 123–138, New York, NY, USA, 2011. ACM.

[24] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 253–267, New York, NY, USA, 2003. ACM.