# Chapter 4

# Authenticated Agreement

Byzantine nodes are able to lie about their inputs as well as received messages. Can we detect certain lies and limit the power of byzantine nodes? Possibly, the authenticity of messages may be validated using signatures?

## 4.1 Agreement with Authentication

**Definition 4.1** (Signature). *Every node can **sign** its messages in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message* `msg`$(x)$ *signed by node $u$ with* `msg`$(x)_u$.

**Remarks:**

- Algorithm 4.2 shows a synchronous agreement protocol for binary inputs relying on signatures. We assume there is a designated "primary" node $p$ that all other nodes know. The goal is to decide on $p$'s value.

**Theorem 4.3.** *Algorithm 4.2 can tolerate $f < n$ byzantine failures while terminating in $f + 1$ rounds.*

*Proof.* Assuming that the primary $p$ is not byzantine and its input is 1, then $p$ broadcasts `value`$(1)_p$ in the first round, which will trigger all correct nodes to decide on 1. If $p$'s input is 0, there is no signed message `value`$(1)_p$, and no node can decide on 1.

If primary $p$ is byzantine, we need all correct nodes to decide on the same value for the algorithm to be correct.

Assume $i < f + 1$ is minimal among all rounds in which any correct node $u$ decides on 1. In this case, $u$ has a set $S$ of at least $i$ messages from other nodes for value 1 in round $i$, including one of $p$. Therefore, in round $i + 1 \leq f + 1$, all other correct nodes will receive $S$ and $u$'s message for value 1 and thus decide on 1 too.

Now assume that $i = f + 1$ is minimal among all rounds in which a correct node $u$ decides for 1. Thus $u$ must have received $f + 1$ messages for value 1, one of which must be from a correct node since there are only $f$ byzantine nodes.

**Algorithm 4.2** Byzantine Agreement with Authentication

```
Code for primary p:
if input is 1 then
    broadcast value(1)_p
    decide 1 and terminate
else
    decide 0 and terminate
end if

Code for all other nodes v:
for all rounds $i \in \{1, \dots, f+1\}$ do
    $S$ is the set of accepted messages value(1)_u.
    if $|S| \geq i$ and value(1)_p $\in S$ then
        broadcast $S \cup \{$value(1)_v$\}$
        decide 1 and terminate
    end if
end for
decide 0 and terminate
```

In this case some other correct node $u'$ must have decided on 1 in some round $j < i$, which contradicts $i$'s minimality; hence this case cannot happen.

Finally, if no correct node decides on 1 by the end of round $f + 1$, then all correct nodes will decide on 0. □

**Remarks:**

- The algorithm only takes $f + 1$ rounds, which is optimal as described in Theorem 3.20.
- Using signatures, Algorithm 4.2 solves consensus for any number of failures! Does this contradict Theorem 3.12? Recall that in the proof of Theorem 3.12 we assumed that a byzantine node can distribute contradictory information about its own input. If messages are signed, correct nodes can detect such behavior – a node $u$ signing two contradicting messages proves to all nodes that node $u$ is byzantine.
- Does Algorithm 4.2 satisfy any of the validity conditions introduced in Section 3.1? No! A byzantine primary can dictate the decision value. Can we modify the algorithm such that the correct-input validity condition is satisfied? Yes! We can run the algorithm in parallel for $2f + 1$ primary nodes. Either 0 or 1 will occur at least $f + 1$ times, which means that one correct process had to have this value in the first place. In this case, we can only handle $f < \frac{n}{2}$ byzantine nodes.
- If the primary is a correct node, Algorithm 4.2 only needs two rounds! Can we make it work with arbitrary inputs? Also, relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?

## 4.2 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is one of the first and perhaps the most instructive protocol for achieving state replication among nodes as in Definition 1.8 with byzantine nodes in an asynchronous network. We present a very simple version of it without any optimizations.

**Definition 4.4** (System Model). *There are $n = 3f + 1$ nodes and an unbounded number of clients. There are at most $f$ byzantine nodes, and clients can be byzantine as well. The network is asynchronous, and messages have variable delay and can get lost. Clients send requests that correct nodes have to order to achieve state replication.*

The ideas behind PBFT can roughly be summarized as follows:

- Signatures guarantee that every node can determine which node/client generated any given message.
- At any given time, every node will consider one designated node to be the *primary* and the other nodes to be *backups*. Since we are in the variable delay model, requests can arrive at the nodes in different orders. While a primary remains in charge (this timespan corresponds to what is called a *view*), it thus has the function of a serializer (cf. Algorithm 1.9).
- If backups detect faulty behavior in the primary, they start a new view and the next node in round-robin order becomes primary. This is called a *view change*.
- After a view change, a correct new primary makes sure that no two correct nodes execute requests in different orders. Exchanging information will enable backups to determine if the new primary acts in a byzantine fashion.

**Definition 4.5** (View). *A **view** is represented locally at each node $i$ by a non-negative integer $v$ (we say $i$ **is in view** $v$) that is incremented whenever the node changes to a different view.*

**Definition 4.6** (Primary; Backups). *A node that is in view $v$ considers node $v \bmod n$ to be the **primary** and all other nodes to be **backups**.*

**Definition 4.7** (Sequence Number). *During a view, a node relies on the primary to pick consecutive integers as **sequence numbers** that function as indices in the global order (cf. Definition 1.8) for the requests that clients send.*

**Remarks:**

- All nodes start out in view 0 and can potentially be in different views (i.e. have different local values for $v$) at any given time.
- The protocol will guarantee that once a correct node has executed a request $r$ with sequence number $s$, then eventually all correct nodes will execute $r$ with sequence number $s$, and that no correct node will execute any $r' \neq r$ with sequence number $s$.

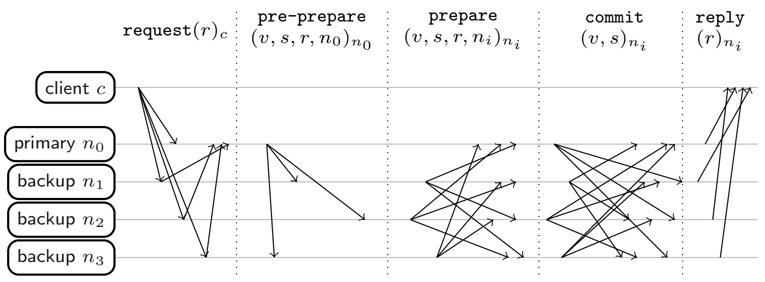

Figure 4.10: The agreement protocol used in PBFT for processing a client request exemplified for a system with 4 nodes. Node $n_0$ is the primary in current view $v$. Time runs from left to right. Messages sent at the same time need not arrive at the same time.

- Correct primaries choose sequence numbers such that they are *dense*, i.e. if a correct primary proposed $s$ as the sequence number for the last request, then it will use $s+1$ for the next request that it proposes.
- Before a node can safely execute a request $r$ with a sequence number $s$, it will wait until it knows that the decision to execute $r$ with $s$ has been reached and is widely known.
- Informally, nodes will collect confirmation messages by sets of at least $2f + 1$ nodes to guarantee that that information is sufficiently widely distributed.

**Definition 4.8** (Accepted Messages). *A correct node that is in view $v$ will only **accept messages** that it can authenticate, that follow the specification of the protocol, whose components can be validated in the same way, and that also belong to view $v$.*

**Lemma 4.9** (2f+1 Quorum Intersection). *Let $S_1$ with $|S_1| \geq 2f + 1$ and $S_2$ with $|S_2| \geq 2f + 1$ each be sets of nodes. Then there exists a correct node in $S_1 \cap S_2$.*

*Proof.* Let $S_1, S_2$ each be sets of at least $2f + 1$ nodes. There are $3f + 1$ nodes in total, thus due to the pigeonhole principle the intersection $S_1 \cap S_2$ contains at least $f + 1$ nodes. Since there are at most $f$ faulty nodes, $S_1 \cap S_2$ contains at least 1 correct node. □

## 4.3 PBFT: Agreement Protocol

First we describe how PBFT achieves agreement on a unique order of requests within a view.

**Remarks:**

- Figure 4.10 shows how the nodes come to an agreement on a sequence number for a client request. Informally, the protocal has these three steps:

1. The primary sends a `pre-prepare`-message to all backups, informing them that he wants to execute that request with the sequence number specified in the message.
2. Backups send `prepare`-messages to all nodes, informing them that they agree with that suggestion.
3. All nodes send `commit`-messages to all nodes, informing everyone that they have committed to execute the request with that sequence number. They execute the request and inform the client.

- Figure 4.10 shows that all nodes can start each phase at different times.
- To make sure byzantine nodes cannot force the execution of a request, every node waits for a certain number of `prepare`- and `commit`-messages with the correct content before executing the request.
- Definitions 4.11, 4.14, 4.16 specify the agreement protocol formally. Backups run Phases 1 and 2 concurrently.

**Definition 4.11** (PBFT Agreement Protocol Phase 1; Pre-Prepared Primary). *In **phase 1** of the agreement protocol, the nodes execute Algorithm 4.12.*

**Algorithm 4.12** PBFT Agreement Protocol: Phase 1

```
    Code for primary p in view v:
1: accept request$(r)_c$ that originated from client $c$
2: pick next sequence number $s$
3: send pre-prepare$(v, s, r, p)_p$ to all backups
    Code for backup b:
4: accept request$(r)_c$ from client $c$
5: relay request$(r)_c$ to primary $p$
```

**Definition 4.13** (Faulty-Timer). *When backup $b$ accepts request $r$ in Algorithm 4.12 Line 4, $b$ starts a local **faulty-timer** (if the timer is not already running) that will only stop once $b$ executes $r$.*

**Remarks:**

- If the faulty-timer expires, the backup considers the primary faulty and triggers a view change. We explain the view change protocol in Section 4.4.
- We leave out the details regarding for what timespan to set the faulty-timer as they are an optimization with several trade-offs to consider; the interested reader is advised to consult [MC99].

**Definition 4.14** (PBFT Agreement Protocol Phase 2; Pre-prepared Backups). *In phase 2 of the agreement protocol, every backup $b$ executes Algorithm 4.15. Once it has sent the* `prepare`*-message, $b$ has **pre-prepared** $r$ for $(v, s)$.*

**Algorithm 4.15** PBFT Agreement Protocol: Phase 2

```
    Code for backup $b$ in view $v$:
1: accept pre-prepare$(v, s, r, p)_p$
2: if $p$ is primary of view $v$ and $b$ has not yet accepted a pre-prepare-message
   for $(v, s)$ and different $r$ then
3:     send prepare$(v, s, r, b)_b$ to all nodes
4: end if
```

**Definition 4.16** (PBFT Agreement Protocol Phase 3; Prepared- Certificate)**.** *A node $i$ that has pre-prepared a request executes Algorithm 4.17. It waits until it has collected $2f$* `prepare`*-messages (including $i$'s own, if it is a backup) in Line 1. Together with the* `pre-prepare`*-message for $(v, s, r)$, they form a **prepared-certificate**.*

**Algorithm 4.17** PBFT Agreement Protocol: Phase 3

```
    Code for node $i$ that has pre-prepared $r$ for $(v, s)$:
1: wait until $2f$ prepare-messages matching $(v, s, r)$ have been accepted (in-
   cluding $i$'s own message, if it is a backup)
2: send commit$(v, s, i)_i$ to all nodes
3: wait until $2f + 1$ commit-messages (including $i$'s own) matching $(v, s)$ have
   been accepted
4: execute request $r$ once all requests with lower sequence numbers have been
   executed
5: send reply$(r)_i$ to client
```

**Remarks:**

- Note that the agreement protocol can run for multiple requests in parallel. Since we are in the variable delay model and messages can arrive out of order, we thus have to wait in Algorithm 4.17 Line 4 until a request has been executed for all previous sequence numbers.
- The client only considers the request to have been processed once it received $f + 1$ `reply`-messages sent by the nodes in Algorithm 4.17 Line 5. Since a correct node only sends a `reply`-message once it executed the request, with $f + 1$ `reply`-messages the client can be certain that the request was executed by a correct node.
- We will see in Section 4.4 that PBFT guarantees that once a single correct node executed the request, then all correct nodes will eventually execute the request with the same sequence number. Thus, knowing that a single correct node executed a request is enough for the client.
- If the client does not receive at least $f+1$ `reply`-messages fast enough, it can start over by resending the request to initiate Algorithm 4.12 again. To prevent correct nodes that already executed the request

from executing it a second time, clients can mark their requests with some kind of unique identifiers like a local timestamp. Correct nodes can then react to each request that is resent by a client as required by PBFT, and they can decide if they still need to execute a given request or have already done so before.

**Lemma 4.18** (Unique Sequence Numbers within View)**.** *If a node gathers a prepared-certificate for $(v, s, r)$, then no node can gather a prepared-certificate for $(v, s, r')$ with $r' \neq r$.*

*Proof.* Assume two (not necessarily distinct) nodes gather prepared-certificates for $(v, s, r)$ and $(v, s, r')$. Since a prepared-certificate contains $2f + 1$ messages, a correct node sent a `pre-prepare`- or `prepare`-message for each of $(v, s, r)$ and $(v, s, r')$ due to Lemma 4.9. A correct primary only sends a single `pre-prepare`-message for each $(v, s)$, see Algorithm 4.12 Lines 2 and 3. A correct backup only sends a single `prepare`-message for each $(v, s)$, see Algorithm 4.15 Lines 2 and 3. Thus, $r' = r$. □

**Remarks:**

- Due to Lemma 4.18, once a node has a prepared-certificate for $(v, s, r)$, no correct node will execute some $r' \neq r$ with sequence number $s$ during view $v$ because correct nodes wait for a prepared-certificate before executing a request (cf. Algorithm 4.17).
- However, that is not yet enough to make sure that no $r' \neq r$ will be executed by a correct node with sequence number $s$ during some later view $v' > v$. How can we make sure that that does not happen?

## 4.4 PBFT: View Change Protocol

If the primary is faulty, the system has to perform a view change to move to the next primary so the system can make progress. Nodes use their faulty-timer (and only that!) to decide whether they consider the primary to be faulty (cf. Definition 4.13).

**Remarks:**

- During a view change, the protocol has to guarantee that requests that have already been executed by some correct nodes will be executed with the same sequence numbers by all correct nodes.
- How can we guarantee that this happens?

**Definition 4.19** (PBFT: View Change Protocol)**.** *In the view change protocol, a node whose faulty-timer has expired enters the **view change phase** by running Algorithm 4.22. During the **new view phase** (which all nodes continually listen for), the primary of the next view runs Algorithm 4.23 while all other nodes run Algorithm 4.24.*

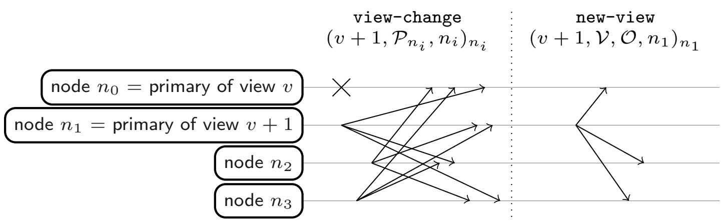
view-change
$(v+1, \mathcal{P}_{n_i}, n_i)_{n_i}$
new-view
$(v+1, \mathcal{V}, \mathcal{O}, n_1)_{n_1}$
node $n_0$ = primary of view $v$
node $n_1$ = primary of view $v+1$
node $n_2$
node $n_3$

Figure 4.20: Node $n_0$ is the primary of current view $v$, node $n_1$ the primary of view $v+1$. Once backups consider $n_0$ to be faulty, they start the view change protocol (cf. Algorithms 4.22, 4.23, 4.24). The X signifies that $n_0$ is faulty.

**Remarks:**

- The idea behind the view change protocol is this: during the view change protocol, the new primary gathers prepared-certificates from $2f+1$ nodes, so for every request that some correct node executed, the new primary will have at least one prepared-certificate.
- After gathering that information, the primary distributes it and tells all backups which requests need to be to executed with which sequence numbers.
- Backups can check whether the new primary makes the decisions required by the protocol, and if it does not, then the new primary must be byzantine and the backups can directly move to the next view change.

**Definition 4.21** (New-View-Certificate)**.** $2f+1$ `view-change`*-messages for the same view v form a **new-view-certificate**.*

**Algorithm 4.22** PBFT View Change Protocol: View Change Phase

```
    Code for backup $b$ in view $v$ whose faulty-timer has expired:
1: stop accepting pre-prepare/prepare/commit-messages for $v$
2: let $\mathcal{P}_i$ be the set of all prepared-certificates that $b$ has collected since the
   system was started
3: send view-change$(v+1, \mathcal{P}_i, i)_i$ to all nodes
```

**Remarks:**

- It is possible that $\mathcal{V}$ contains a prepared-certificate for a sequence number $s$ while it does not contain one for some sequence number $s' < s$. For each such sequence number $s'$, we fill up $\mathcal{O}$ in Algorithm 4.23 Line 4 with `null`-requests, i.e. requests that backups understand to mean "do not do anything here".

**Theorem 4.25** (Unique Sequence Numbers across Views)**.** *Together, the agreement protocol and the view change protocol guarantee that if a correct node executes a request $r$ in view $v$ with sequence number $s$, then no correct node will execute any $r' \neq r$ with sequence number $s$ in any view $v' \geq v$.*

**Algorithm 4.23** PBFT View Change Protocol: New View Phase - Primary

```
    Code for primary $p$ of view $v+1$:
 1: accept $2f+1$ view-change-messages (including possibly $p$'s own) in a set
    $\mathcal{V}$ (this is the new-view-certificate)
 2: let $\mathcal{O}$ be a set of pre-prepare$(v+1, s, r, p)_p$ for all pairs $(s, r)$ where at
    least one prepared-certificate for $(s, r)$ exists in $\mathcal{V}$
 3: let $s^{\mathcal{V}}_{max}$ be the highest sequence number for which $\mathcal{O}$ contains a
    pre-prepare-message
 4: add to $\mathcal{O}$ a message pre-prepare$(v+1, s', \texttt{null}, p)_p$ for every sequence
    number $s' < s^{\mathcal{V}}_{max}$ for which $\mathcal{O}$ does not yet contain a pre-prepare-message

 5: send new-view$(v+1, \mathcal{V}, \mathcal{O}, p)_p$ to all nodes
 6: start processing requests for view $v+1$ according to Algorithm 4.12 starting
    from sequence number $s^{\mathcal{V}}_{max}+1$
```

**Algorithm 4.24** PBFT View Change Protocol: New View Phase - Backup

```
    Code for backup $b$ of view $v+1$ if $b$'s local view is $v' < v+1$:
 1: accept new-view$(v+1, \mathcal{V}, \mathcal{O}, p)_p$
 2: stop accepting pre-prepare-/prepare-/commit-messages for $v$// in case
    $b$ has not run Algorithm 4.22 for $v+1$ yet
 3: set local view to $v+1$
 4: if $p$ is primary of $v+1$ then
 5:   if $\mathcal{O}$ was correctly constructed from $\mathcal{V}$ according to Algorithm 4.23 Lines 2
      and 4 then
 6:     respond to all pre-prepare-messages in $\mathcal{O}$ as in normal case operation,
        starting from Algorithm 4.15
 7:     start accepting messages for view $v+1$
 8:   else
 9:     trigger view change to $v+2$ using Algorithm 4.22
10:   end if
11: end if
```

*Proof.* If no view change takes place, then Lemma 4.18 proves the statement. Therefore, assume that a view change takes place, and consider view $v' > v$. We will show that if some correct node executed a request $r$ with sequence number $s$ during $v$, then a correct primary will send a `pre-prepare`-message matching $(v', s, r)$ in the $\mathcal{O}$-component of the `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message. This guarantees that correct nodes will eventually agree on $(s, r)$.

Consider the new-view-certificate $\mathcal{V}$ (see Algorithm 4.23 Line 1). If any correct node executed request $r$ with sequence number $s$, then due to Algorithm 4.17 Line 3, there is a set $R_1$ of at least $2f+1$ nodes that sent a `commit`-message matching $(s, r)$, and thus the correct nodes in $R_1$ all collected a prepared-certificate in Algorithm 4.17 Line 1.

The new-view certificate contains `view-change`-messages from a set $R_2$ of $2f+1$ nodes. Thus according to Lemma 4.9, there is at least one correct node

$c_r \in R_1 \cap R_2$ that both collected a prepared-certificate matching $(s, r)$ and whose `view-change`-message is contained in $\mathcal{V}$.

Therefore, if $c_r$ executed $r$ with sequence number $s$, then $\mathcal{V}$ contains a prepared-certificate matching $(s, r)$ from $c_r$. Thus, if $c_r$ executed $r$ with sequence number $s$, then due to Algorithm 4.23 Line 2, a correct primary $p$ sends a `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message where $\mathcal{O}$ contains a `pre-prepare`$(v', s, r, p)$-message.

Correct backups will enter view $v'$ only if the `new-view`-message for $v'$ contains a valid new-view-certificate $\mathcal{V}$ and if $\mathcal{O}$ was constructed correctly from $\mathcal{V}$, see Algorithm 4.24 Line 5. They will then respond to the messages in $\mathcal{O}$ before they start accepting other `pre-prepare`-messages for $v'$ due to the order of Algorithm 4.24 Lines 6. Therefore, for the sequence numbers that appear in $\mathcal{O}$, correct backups will only send `prepare`-messages responding to the `pre-prepare`-messages found in $\mathcal{O}$ due to Algorithm 4.15 Lines 2 and 3. This guarantees that in $v'$, for every sequence number $s$ that appears in $\mathcal{O}$, backups can only collect prepared-certificates for the triple $(v', s, r)$ that appears in $\mathcal{O}$.

Together with the above, this proves that if some correct node executed request $r$ with sequence number $s$ in $v$, then no node will be able to collect a prepared-certificate for some $r' \neq r$ with sequence number $s$ in any view $v' \geq v$, and thus no correct node will execute $r'$ with sequence number $s$. $\square$

**Remarks:**

- A faulty new primary could delay the system indefinitely by never sending a `new-view`-message. To prevent this, as soon as a node sends its `view-change`-message for $v + 1$, it starts its faulty-timer and stops it once it accepts a `new-view`-message for $v + 1$. If the timer runs out before being stopped, the node triggers another view change.

- Since at most $f$ consecutive primaries can be faulty, the system makes progress after at most $f + 1$ view changes.

- We described a simplified version of PBFT; any practically relevant variant makes adjustments to what we presented. The references found in the chapter notes can be consulted for details that we did not include.

## Chapter Notes

PBFT is perhaps the central protocol for asynchronous byzantine state replication. The seminal first publication about it, of which we presented a simplified version, can be found in [MC99]. The canonical work about most versions of PBFT is Miguel Castro's PhD dissertation [Cas01].

Notice that the sets $\mathcal{P}_i$ in Algorithm 4.22 grow with each view change as the system keeps running since they contain all prepared-certificates that nodes have collected so far. All variants of the protocol found in the literature introduce regular *checkpoints* where nodes agree that enough nodes executed all requests up to a certain sequence number so they can continuously garbage-collect prepared-certificates. We left this out for conciseness.

Remember that all messages are signed. Generating signatures is somewhat pricy, and variants of PBFT exist that use the cheaper, but less powerful Message Authentication Codes (MACs). These variants are more complicated because MACs only provide authentication between the two endpoints of a message and cannot prove to a third party who created a message. An extensive treatment of a variant that uses MACs can be found in [CL02].

The publication of PBFT started an "arms race" that lead to the development of numerous asynchronous byzantine state replication protocols. Other important protocols are Q/U [AEMGG+05], HQ [CML+06], and Zyzzyva [KAD+07]. An overview over the relevant literature can be found in [**?** ].

This chapter was written in collaboration with Georg Bachmeier.

## Bibliography

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[Cas01] Miguel Castro. *Practical Byzantine Fault Tolerance*. Ph.d., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[MC99] Barbara Liskov Miguel Castro. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.