The Case for Byzantine Fault Detection

Andreas Haeberlen
Rice University, MPI-SWS

Petr Kouznetsov
MPI-SWS

Peter Druschel
MPI-SWS

Technical Report MPI-SWS-2006-001

1 Introduction

Distributed systems are subject to a variety of failures and attacks. In this paper, we consider general (Byzantine) failures [11], in which a failed node may exhibit arbitrary behavior. In particular, a failed node may corrupt its local state, send random messages, or even send specific messages aimed at subverting the system. Many security attacks can be modeled as Byzantine failures, such as censorship, freeloading, misrouting, or data corruption.

Systems can be protected with Byzantine fault tolerance (BFT) techniques, which can *mask* a bounded number of Byzantine failures, e.g. using state machine replication [4]. BFT is a very powerful technique, but it has its costs. In a practical system that needs to tolerate up to f concurrent Byzantine failures, BFT cannot be implemented with less than $3f + 1$ replicas [3]. Moreover, BFT scales poorly to large replica groups; as more servers are added, the throughput of the system may actually decrease [7].

In this paper, we explore an alternative approach that aims at *detecting* rather than masking faulty behavior. In this approach, the system does not make any attempt to hide the symptoms of Byzantine faults. Rather, each node is equipped with a detector that monitors the other nodes for signs of faulty behavior. If the detector determines that another node has become faulty, it notifies the local node, which can then take appropriate action. For example, it can cease to communicate with the faulty node; once all correct nodes have followed suit, the faulty node is isolated and the fault is contained.

Specifically, we consider detection systems that are based on *accountability* [15]. With accountability, each action is associated with the identity of the node that has taken it, which allows the system to gather irrefutable *evidence* of faulty behavior. This has two important advantages: First, nodes can use the evidence to convince other nodes that a fault has occurred. Second, the evidence enables the system to resolve he-said-she-said situations in which two nodes accuse each other of having failed.

Our goals in this paper are threefold: First, we examine the trade-offs between fault detection and traditional BFT. Second, we give a precise definition of the class of Byzantine faults that can be detected with this approach. Finally, we give a brief sketch of a practical system that implements such a detector.

1.1 The case for fault detection

Clearly, techniques that mask Byzantine failures are easy to use because, unlike fault detection systems, they provide the application designer with the abstraction of a system in which failures simply do not occur. So what reasons are there to opt for fault detection?

You need fewer machines. If a system can suffer up to f concurrent failures, BFT cannot be implemented with less than $3f + 1$ machines. Detection, on the other hand, can be accomplished even with a single correct machine; hence, it requires only $f + 1$ machines[1]. The importance of this result is not only in the reduced hardware requirement. BFT is useful only if node failures are not correlated. Thus, all machines should ideally run different operating systems, different application software, have separate power supplies, etc., to ensure they do not have any common vulnerabilities. This is easier to accomplish for a smaller number of machines.

You can tolerate more failures. With BFT, the *fraction* of machines in the system that are faulty must be below 33% at all times. In detection-based systems, faults can be detected as long as there is one correct node in the system, irrespective of the number of faulty nodes. Thus, it can be used in environments where a high fraction of the nodes (say, 90%) can fail simultaneously.

You can provision for the common case. In a BFT system, all replicas must process each request promptly,

[1] This does not contradict the impossibility results for agreement [3] because detection systems do not guarantee safety.

since the client cannot make progress before most of them have responded. In a detection-based system, however, a *single* replica can process each request and respond immediately; the other replicas can later check the response during a period of light load. Hence, a BFT system must be provisioned such that each machine can handle the peak load, while in a detection system, each machine must merely be able to handle the *average* load.

Detection is cheaper. The reason is that asynchronous checking avoids the consensus required in state-machine replication, and it enables the aggregation of messages, state and processing associated with detection. Also, there is no need for the replicas to be strongly consistent, which makes it much easier to handle view changes.

1.2 Uses of fault detectors

We consider a fully distributed detection system where every node is equipped with its own detector, which watches for faults on the other nodes. Once this detector reports a fault, the local node can respond in various ways. As a first step, it can stop communicating with the faulty node. The node can then distribute the evidence in its possession to other nodes, so they can also respond and thus isolate the faulty node. Finally, the node can initiate recovery. For example, a storage system can create additional replicas of all objects stored on the faulty node and/or notify a human operator, who can then repair the faulty node.

The mere presence of a detection system can reduce the likelihood of certain faults. For example, it can discourage attackers and freeloaders by creating a disincentive to cheating, since a faulty node risks isolation and expulsion from the system. Furthermore, if the system maintains a binding from node identifiers to real-world principals, then even the owner of a faulty node could be exposed and held legally responsible.

Fault detection also has its limits. Detection is not sufficient for failures that have irreversible and serious effects, such as deleting all copies of an important document. However, detection and accountability offers an efficient and scalable alternative to BFT for a large class of real-world failures, including freeloading, censorship, and denial-of-service. Moreover, we believe that detection can be used in combination with BFT (e.g. to prevent BFT from reaching its failure bound by ejecting faulty nodes), which would allow the design of dependable, yet scalable distributed systems. The wide-spread reliance on accountability in human society, both to discourage unwanted behavior and to encourage compliance with the law, provides further justification for the approach.

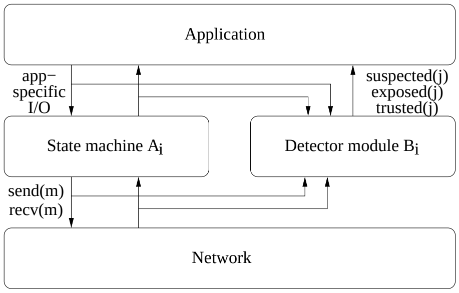

Figure 1: Information flow between application, protocol, and detector module on node i

2 Detectable failures

A perfect detection system would immediately detect any Byzantine fault. The power of a practical, efficient detection system, however, is necessarily limited. In this paper, we will assume that the detector on a correct node can observe all messages sent and received by that node. This clearly means that some Byzantine faults are not observable and therefore *cannot* be detected. For example, a faulty storage node might report that it is out of disk space, which cannot be verified without knowing the actual state of its disks.

In the following, we formally define the class of Byzantine faults that can be detected under this assumption. We distinguish two types of faults. Informally, a node i is *detectably faulty* if the behavior it exposes to correct nodes could not be observed if i were correct, and *detectably ignorant* if i ignores a message sent to i by a correct node. For example, if a correct node requests some service that i is supposed to grant, i is detectably faulty if it denies the request, and detectably ignorant if it pretends that it has not received the request at all.

2.1 System model

We consider a set Π of *nodes*. Every node i is modeled as a state machine A_i and a detector module B_i (Figure 1). Informally, we say that a node i is correct if it respects the specifications of both A_i and B_i. Otherwise, the node is faulty.

Nodes communicate with each other through message passing. We assume that messages are uniquely identified. For a message m, let *sender*(m) and *receiver*(m) denote the sender and the receiver of m, respectively. For the moment, we do not put any restrictions on local processing time and communication delays. However, we assume that, after some number of retransmissions, a message sent from a correct node to a correct node is eventually received.

An *event* is either *send*$_i(m) \in O_i$, where $i =$ *sender*(m), or *receive*$_j(m) \in I_j$, where $j =$

receiver(m), or an application-specific input or output.

An *execution* E is a sequence of events such that in E, each m is sent and received at most once, and each *receive*$_i(m)$ is preceded by the corresponding *send*$_j(m)$. We distinguish events associated with the state machine A_i and events associated with the detector module B_i. $E|A_i$ denotes the subsequence of E that consists of all events associated with A_i in E, and $E|B_i$ denotes the subsequence of E that consists of all events associated with B_i in E. We say that a node i is *correct* in E if (1) $E|A_i$ (respectively, $E|B_i$) *conforms* to A_i (respectively, B_i), i.e., if the sequence of outputs produced in $E|A_i$ ($E|B_i$) is legal, given A_i (B_i) and the sequence of inputs in $E|A_i$ ($E|B_i$), and (2) if E is infinite, then both $E|A_i$ and $E|B_i$ are also infinite. Otherwise we say that i is *faulty* in E.

2.2 Detectable faultiness and ignorance

We define a *history* of a node i as a sequence of events of A_i. A history h of a node i is *valid* if it conforms to A_i, i.e. if, given the sequence of incoming messages and application-specific inputs in h, A_i could have produced the sequence of outgoing messages and application-specific outputs in h. A pair (h_1, h_2) of histories of i is *consistent* if h_1 is a prefix of h_2, or vice versa. If i is a correct node, one trivial example of a valid history is $E|A_i$.

Let $\mathcal{M}(E)$ denote the set of messages received by the nodes in an execution E. We assume that there exists a *history map* φ that associates every message $m \in \mathcal{M}(E)$ with a history of *sender*(m). For a correct node, $\varphi(m)$ is the prefix of the local execution $E|$*sender*(m) up to and including *send*(m). Thus, for any message m sent by a correct node, $\varphi(m)$ is valid, and for every pair of messages m and m' sent by a correct node, $\varphi(m)$ and $\varphi(m')$ are consistent.

We say that a message m is *observable in* E if there exists a correct node i and a sequence of messages $m_1, \ldots, m_k$ such that

(i) $m_1 = m$,

(ii) *receive*(m_k) belongs to $E|A_i$,

(iii) for all $j = 2, \ldots, k$: *receive*(m_{j-1}) belongs to $\varphi(m_j)$.

In other words, m is observable if it causally precedes at least one event on a correct node.

We say that a node i is *detectably faulty* with respect to a message m in an execution E if m that was sent by i, is observable in E, and satisfies one of the following properties:

(1) $\varphi(m)$ is not valid (for i)

(2) There exists a message m' that was also sent by i and is observable in E, such that $\varphi(m)$ is inconsistent with $\varphi(m')$

The set of nodes causally affected by m and m' (if m' exists) are called *accomplices* of i with respect to m.

We say that a node i is *detectably ignorant* in E if i is not detectably faulty in E and there exists a message m sent to i by a correct node, such that, for all observable messages m' sent by i, *receive*$_i(m)$ does not appear in $\varphi(m')$.

2.3 Guarantees

When the detector module B_i on a correct node i has seen evidence of faulty behavior on another node j, it sends a *failure indication* to its local application process. We define three different types of indications: $trusted_j$, $suspected_j$ and $exposed_j$. Intuitively, if the module B_i outputs $suspected_j$, there is evidence that j is ignoring certain inputs, e.g. by refusing to accept a service request from a correct node. If it outputs $exposed_j$, there exists a *proof* that j is faulty, i.e. that it has deviated from the specification of its state machine A_j. Finally, B_i outputs $trusted_j$ while none of the other conditions hold.

We can use a definition similar to that of [5, 9] to describe these properties. Thus, the detection system guarantees that the following properties hold in every execution:

- **Eventual strong completeness:** (1) Eventually, every detectably ignorant node is suspected forever by every correct node, and (2) if a node i is detectably faulty with respect to a message m, then eventually, some faulty accomplice of i (with respect to m) is exposed or forever suspected by every correct node.
- **Eventual strong accuracy:** (1) No correct node is forever suspected by a correct node, and (2) no correct node is ever exposed by a correct node.

Note that the detector need not guarantee that a correct node is always trusted by another correct node; it can jump from *trusted* to *suspected* and back, e.g. due to long message delays. Further, a detectably faulty node might never be detected. However, if a node is detectably faulty, then *some* faulty node will eventually be exposed or suspected forever. Thus, if there are only finitely many faulty nodes in the system, correct nodes can be affected by their behavior only finitely long.

3 A practical detector for Byzantine faults

To show that detection systems are practical, we now briefly sketch the design of PeerReview, a system that can provide the guarantees stated in Section 2.3. A proof can be found in Appendix A. We have implemented PeerReview and initial results suggest that it is practical and efficient. An experimental evaluation is the subject of a future, full paper.

3.1 Assumptions and goals

For PeerReview, we assume that the system can be modeled as described in Section 2.1, with two additional assumptions: First, that the protocol is *deterministic*, i.e. produces the same outputs given the same sequence of inputs. This is a fairly common assumption in state machine replication [4, 10]. Second, that nodes have *strong identities* and a keypair that can be used to sign messages. This can be accomplished, for instance, by giving each node an identity certificate, signed by a certification authority, that ties its public key to its node identifier.

We also make the common assumption that the attacker does not have the ability to break cryptographic signatures. Other than that, the Byzantine nodes may behave arbitrarily and/or collude with each other.

3.2 Secure histories and commitment

Each node is required to keep a log of all the inputs and outputs of its local state machine A_i. The log is organized as a hash chain, similar to a secure history [13], such that the top-level hash covers the contents of the entire log. Furthermore, each node must frequently *commit* to the contents of its log by publishing an *authenticator*, i.e. a signed copy of its top-level hash value. This makes the log *tamper-evident* and ensures that nodes cannot revise history [13].

Nodes must sign all messages they send, and acknowledge all messages they receive. If a message is not acknowledged after several retries, it is broadcast to the other nodes, who then challenge the node to accept the message. This ensures that a node is suspected by all correct nodes if it refuses to accept a message.

Each message or acknowledgment m contains an authenticator, as well as a short proof that $send(m)$ or $receive(m)$ was the top-level entry of the corresponding log. The recipient extracts the authenticators and, once in a while, forwards them to the other nodes. Thus, all nodes are eventually made aware of all authenticators that have been sent to a correct node.

3.3 Auditing

Each node i is periodically *audited* by every other node[2]. During an audit, the auditor j first asks i for a signed log segment that covers all entries since the last audit. j then validates the log against the most current authenticator it has obtained for i. If i refuses to comply, j begins to suspect i.

Next, j performs a *consistency check* to see if the log matches all the recent authenticators it has obtained for i. If this fails, i has forked its log or is keeping multiple copies, and j obtains a signed confession. The evidence is then made available to other nodes, who can thus mark i as exposed.

In a third step, j extracts all authenticators from the log segment and forwards them to the other nodes. This ensures that, even if i is faulty and has not performed this step earlier, the other nodes will eventually be aware of all relevant authenticators. This step requires $O(N^2)$ messages; however, these messages are small and can be heavily aggregated.

Finally, j performs a *conformance check*. It instantiates a local copy of the state machine A_i and initializes it with a recent checkpoint from the log. Then it replays all the inputs from the log and checks whether the corresponding outputs match the ones in the log. Thus, j can check protocol conformance without an explicit protocol specification. If it detects a divergence, it has obtained a signed confession and can thus expose i.

3.4 Checking evidence

If a node j detects a fault on a node i, it obtains one of two types of evidence. If i is detectably faulty, j obtains either a) an authenticator and a log, both of which are signed but do not match, or b) a signed log segment that fails the conformance check. Both constitute a signed confession. If i is detectably ignorant, j obtains a challenge (e.g. a request for a certain log segment) that i cannot answer, except by providing a signed confession.

Both types of evidence can be distributed to the other nodes, who can verify them independently, either by repeating the checks performed by j (in case of a signed confession) or by contacting i and checking its response (in case of a challenge). PeerReview ensures that this check will always fail for a correct node, since they never generate signed confessions and can respond to any challenge.

The output of the PeerReview failure detector on a given node is reliable if, and only if, the node has a valid copy of the state machine to be run by all the nodes in

[2] We assume here that there are exactly $f + 1$ nodes in the system. In larger systems, it is sufficient that $f + 1$ other nodes audit a given node, but we omit the details due to lack of space.

the system. A node can ensure this, for instance, by obtaining a signed binary program from a trusted authority.

To bound the space required for logs, nodes may be allowed to discard old log entries, e.g. after a month. In this case, older evidence can no longer be verified and must be discarded as well, which eventually allows faulty nodes to return to the system. This is acceptable, as long as the system has ample time to respond to the failure and initiate repair.

3.5 Discussion

PeerReview provides the benefits of detection we outlined in Section 1.1. Even a single correct node can, through auditing, detect and obtain evidence of any observable faults, and it can take appropriate action. Auditing can be performed asynchronously, so the nodes can defer the corresponding overhead to periods of light load. It requires comparatively few messages: With a failure bound of f nodes, Castro and Liskov's BFT protocol [4] requires $18f^2+9f+2$ messages per request, while PeerReview uses only f^2+3f+2. Moreover, most of these messages can be aggregated.

4 Related work

Our concept of a detection system is based on the failure detectors by Chandra and Toueg [5]. These were defined for crash failures, but Malkhi and Reiter [12] later extended them to the Byzantine failure model. Kihlstrom et al. [9] have introduced several classes of failure detectors that expose *detectable* Byzantine failures. However, they consider classes of algorithms in which all messages are broadcast, and in which processes know when to expect messages from other processes. PeerReview does not require these assumptions.

State machine replication [10, 14] is a classical technique for masking a limited number of Byzantine faults. Today's state-of-the-art BFT techniques, e.g. [4], are based on this idea. The BAR model [1] combines this approach with a system structure that causes the system to operate in a Nash equilibrium. Thus, BAR can additionally tolerate an unbounded number of rational nodes that are willing to deviate from the protocol in order to increase their own utility. Although both techniques are related to detection systems, neither can identify faulty nodes, and both require more resources.

Alvisi et al. [2] introduced a technique that monitors quorum systems and raises an alarm if the failure assumptions are about to be violated. This technique is probabilistic and, unlike PeerReview, cannot identify which nodes are faulty.

Intrusion detection systems [6] can detect certain types of protocol violations; however, unlike PeerReview, the heuristics used in IDS tend to produce either false positives, false negatives, or both. Reputation systems such as EigenTrust [8] can be used against Byzantine failures, but, unlike PeerReview, they cannot prevent a coalition of malicious nodes from denouncing a correct node. Finally, trusted computing platforms like TCG/Palladium can detect failures that involve software modifications, but require special hardware and force users to give up some control over their own equipment. PeerReview works on commodity hardware and merely checks protocol conformance.

5 Conclusion and future work

In this paper, we have discussed an alternative approach to handling Byzantine faults, in which the system does not mask faults but rather detects and responds to them. We have formally specified the class of faults that can be detected with this approach, and we have sketched the design of a practical system that implements it. To our knowledge, this is the first practical, general-purpose algorithm for detecting Byzantine faults.

We believe that this work opens up a new and interesting direction for future research. Detection can be used to protect a much wider range of systems against Byzantine faults, especially where deploying BFT is infeasible or prohibitively expensive. For example, large-scale distributed systems have long been suffering from freeloading and various attacks. Detection could provide accountability and thus an inexpensive yet highly effective defense.

Also, we believe that further research in detection systems will yield a variety of new detectors with interesting tradeoffs. For example, more powerful detectors could be constructed by adding more sensors, such as attestation, and hybrids between detection and BFT could allow more fine-grained tradeoffs between protection and overhead.

References

[1] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proceedings of SOSP'05*, Oct 2005.

[2] L. Alvisi, D. Malkhi, E. Pierce, and M. Reiter. Fault detection for Byzantine quorum systems. In *Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, pages 357–371, Jan 1999.

[3] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1995.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of OSDI'99*, pages 173–186, 1999.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[6] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[7] J. R. Douceur and J. Howell. Byzantine fault isolation in the Farsite distributed file system. In *Proc. of IPTPS'06*, Feb 2006.

[8] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in p2p networks. In *Proc. 12th International WWW Conference*, May 2003.

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[10] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Prog. Lang. Syst.*, 6(2):254–280, 1984.

[11] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[12] D. Malkhi and M. K. Reiter. Unreliable intrusion detection in distributed computations. In *CSFW*, pages 116–125, 1997.

[13] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Jan 2002.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[15] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the 1st Workshop on Hot Topics in System Dependability*, Jun 2005.

A Proof

Our goal is to prove that PeerReview has the properties defined in Section 2.3 of this paper, namely:

- **Eventual strong completeness:** (1) Eventually, every detectably ignorant node is suspected forever by every correct node, and (2) if a node i is detectably faulty with respect to a message m, then eventually, some faulty accomplice of i (with respect to m) is exposed or forever suspected by every correct node.
- **Eventual strong accuracy:** (1) No correct node is forever suspected by a correct node, and (2) no correct node is ever exposed by a correct node.

We begin by giving some additional details of the PeerReview algorithm in Section A.1. Then we prove each of the above four claims in Sections A.2 and A.3.

A.1 Validating evidence

In the context of PeerReview, *evidence* is a piece of data that supports a statement about the behavior of a particular node. There are three different kinds of evidence for a node i. A *challenge* `chal(x)` states that i has refused to respond to request `x` and therefore may be detectably faulty or detectably ignorant. A *response* `resp(x)` states that i has in fact responded to request `x` and thus refutes the corresponding challenge. Finally, a *proof* `proof(x)` states that i is detectably faulty. A proof also includes some information about the nature of the fault.

A faulty node could try to incriminate a correct node by forging evidence. Therefore, every node must establish that the evidence is *valid* before using it. A list of all types of evidence about a node i, including the conditions under which they are valid, is given below:

- `chal(audit,`$a_{i,x}$`,`$a_{i,y}$`)` is evidence that node i is refusing to return the log segment $\{e_x, \ldots, e_y\}$. The challenge is valid iff both $a_{i,x}$ and $a_{i,y}$ are authenticators signed by i, and $y > x$.
- `resp(audit,`$a_{i,x}$`,`$a_{i,y}$`,`L`)` shows that i has properly responded to the challenge above. The response is valid iff `chal(audit,`$a_{i,x}$`,`$a_{i,y}$`)` is valid, $L = \{e_x, \ldots, e_y\}$ is a well-formed log segment signed by i, and the hashes in $a_{i,x}$ and $a_{i,y}$ match those of the corresponding entries in L.
- `chal(send,`m`)` is evidence that i is refusing to accept a message m. The challenge is valid iff $receiver(m) = i$.
- `resp(send,`m`,`$a_{i,x-1}$`,`$a_{i,x}$`)` shows that i has in fact accepted the message m. The response is valid iff `chal(send,`m`)` is valid, both $a_{i,x-1}$ and $a_{i,x}$ are signed by i, and $a_{i,x}$ validates[3] a log entry $send_i(ack(m))$.
- `proof(inconsistent,`$a_{i,x}$`,`L`)` shows that i is maintaining several inconsistent histories. The proof is valid iff L is signed by i and contains an entry e_x that has the same sequence number as $a_{i,x}$, but a different hash.
- `proof(invalid,`c`,`L`)` shows that i's history does not conform to its state machine A_i. The proof is valid iff L is signed by i, c matches the first checkpoint in L, and L fails the conformance check for A_i.

A piece of evidence is *invalid* if it is not valid according to these rules. Invalid evidence is not considered further by PeerReview.

Each node j maintains an *evidence set* ε_{ij} for every other node i. For simplicity, we will assume that if j and k are correct nodes, then ε_{ij} and ε_{ik} are eventually consistent, so we can treat them as a single set ε_i. In practice, this can be achieved e.g. by allowing correct nodes to gossip about evidence.

To save space, nodes immediately discard invalid evidence. Also, if ε_i contains both a valid challenge c and a matching, valid response $r(c)$, the nodes may eventually discard both.

A node j generates failure indications for another node i as follows: If ε_{ij} contains a valid challenge but no matching, valid response, then j outputs $suspected_i$. If ε_{ij} contains a valid proof, then j outputs $exposed_i$. In all other cases, j outputs $trusted_i$.

[3] To check this, the previous hash h_{x-1} is required, which can be taken from $a_{i,x-1}$.

A.2 Eventual strong completeness

Theorem 1 *Eventually, every detectably ignorant node is suspected forever by every correct node.*

Proof: Assume the opposite, i.e. there is a detectably ignorant node i and a correct node k such that, for every time t, there is another time $t' > t$ at which k does not suspect i. Since i is detectably ignorant, we know that it is not detectably faulty, and that there exists some message m that a correct node j has sent to it, but i has never sent another message m' such that $receive_i(m)$ appears in $\varphi(m')$.

Since acknowledgments are mandatory, j must have resent m several times, then it must eventually have given up and added a challenge chal(send,m) to ε_i, so eventually all correct nodes must have started suspecting i. Let t_1 be the time k is first notified of the challenge. By our assumption, we know that there is a time $t_2 > t_1$ at which k does not suspect i. But this is only possible if the challenge has been refuted because resp(send,m,$a_{i,x-1}$,$a_{i,x}$) has been added to ε_i, and because $a_{i,x}$ validates a log entry $send_i(ack(m))$. By definition, i is not detectably faulty, so $receive_i(m)$ must have preceded $send_i(ack(m))$ in $\varphi(ack(m))$, which means that i cannot be detectably ignorant. This is a contradiction. □

The following discussion is related to a mechanism described in Section 3.2. Recall that nodes commit to the contents of their logs by publishing authenticators, or signed hashes of their logs. We say that a node j is *notified* of a history $\varphi(m)$ of another node i if it receives an authenticator $a_{i,x}$ for a log that corresponds to $\varphi(m)$. The authenticator allows j to obtain $\varphi(m)$ from i, which j can then validate against the hash value in the authenticator. If i does not comply, j can use $a_{i,x}$ in an audit challenge and thus cause all correct nodes to suspect i.

As described in Section 3.3, the notification does not have to be performed by i itself. Since an authenticator is included with each log entry $receive_i(m)$, other nodes can extract the authenticators during audits and then perform the notification on i's behalf.

Lemma 1 *If a message m is observable by a correct node c via a chain of messages $m_1, \ldots, m_k$, then either each correct node is notified of $\varphi(m_x)$ for all $1 \leq x \leq k$, or some $sender(m_x)$ is exposed or forever suspected by all correct nodes.*

Proof: Note that, by the definition of observability, $m_1 = m$, $receiver(m_k) = c$, and for all $2 \leq j \leq k$, $receive(m_{j-1})$ belongs to $\varphi(m_j)$.

We begin by observing that, since c is correct, it will certainly notify all other nodes of $\varphi(m_k)$. Thus, it is sufficient if we can show that, if all nodes are notified of $\varphi(m_x)$ (for some $x > 1$), then all nodes are also notified of $\varphi(m_{x-1})$, or $sender(m_x)$ is either exposed or forever suspected.

If $sender(m_x)$ is exposed or forever suspected, the claim follows immediately. Otherwise, it must have fully cooperated with all correct nodes. If it had refused to answer an audit, an audit challenge would eventually have been added to its evidence set, and it would have been suspected forever by all correct nodes. Moreover, we know that each correct node must have seen a *valid* log; otherwise that node would have added an invalid proof of misbehavior the evidence set, and $sender(m_x)$ would have been exposed.

Let h_x be the history of $sender(m_x)$ as observed by some correct node c_x. We know that $\varphi(m_x)$ ends with an entry $send_{receiver(m_x)}(m_x)$ (otherwise $receiver(m_x)$ would never have accepted m_x). Therefore, we know that h_x must also contain $send_{receiver(m_x)}(m_x)$, because otherwise c_x would have added an inconsistent proof of misbehavior to its evidence set, and $sender(m_x)$ would have been exposed. But if h_x contains $send_{receiver(m_x)}(m_x)$, it must also contain an entry $receive(m_{x-1})$ (if this was not the case, m_k would not be causally connected to m_{k-1}). When auditing this entry, c_x extracts $\varphi(m_{x-1})$ and notifies all the other nodes.

The claim follows by reverse induction over x. □

Theorem 2 *If a node i is detectably faulty with respect to some message m, then eventually, either i itself or some faulty accomplice of i with respect to m is exposed or forever suspected by every correct node.*

Proof: Assume the contrary, i.e. that there is a node i that is detectably faulty with respect to some message m, but neither i nor any of its faulty accomplices with respect to m is exposed or forever suspected by some correct node j. Since correct nodes are not exposed or forever suspected under any circumstances (see Section A.3), it follows that *no* node along the path of causality is exposed or forever suspected.

By Lemma 1, we know that under these circumstances, all nodes must eventually be notified of $\varphi(m)$. Let c be some correct node. We know that i has cooperated with c and responded to all of its audits, since c does not forever suspect i. We also know that i's history h, as seen by c, is valid, since c does not expose i. For the same reason, we know that $\varphi(m)$ is consistent with h, so this cannot be the reason i is detectably faulty.

The only other potential reason i could be detectably faulty is that it has sent some other message m' that is observable by some correct node c', such that $\varphi(m)$ is inconsistent with $\varphi(m')$. But, by Lemma 1, we know that either some node along the path of m' is exposed or forever suspected, or all correct nodes are eventually notified of $\varphi(m')$ as well. In the first case, i is faulty with respect to m', and a faulty accomplice is exposed or forever suspected, so the theorem follows. In the second case, we know that c has not exposed i, so $\varphi(m')$ must be consistent with h and therefore also with $\varphi(m)$. This is a contradiction. □

A.3 Eventual strong accuracy

Theorem 3 *No correct node is forever suspected by a correct node.*

Proof: Assume the opposite, i.e. there is a correct node i that is forever suspected by another correct node k after some time t_1. Let $t_2 > t_1$ be the first time k checks ε_i. Since k still suspects i after t_2, ε_i must have contained some valid, unrefuted challenge c. Moreover, we know that k must have challenged i with c, but i did not provide a valid response.

The challenge c can either be `chal(audit,`$a_{i,x}$`,`$a_{i,y}$`)` or `chal(send,`m`)`. If it is an `audit` challenge, it can only be valid if both $a_{i,x}$ and $a_{i,y}$ are authenticators signed by i, and $y > x$ (recall that signatures cannot be forged). But since i is correct, its log is well-formed, so the authenticators $a_{i,x}$ and $a_{i,y}$ must correspond to existing log entries e_x and e_y with the corresponding hash values. Thus, i can extract the log segment $L = e_x, \ldots, e_y$ and use it to construct a valid response.

Now assume c is a `send` challenge. There are two cases: Either i has previously received m, or it has not. If the former holds, i, being correct, must have an entry $send_i(ack(m))$ in its log. Using the authenticator $a_{i,x}$ covering this entry, it can construct a valid response. If i has not yet received m, it can accept it now, which produces the required log entries and again enables i to construct a valid response.

But if i can construct a valid response, it would have done so and replied to k. Eventually, this reply would have been received by k, so it cannot have suspected i forever. This is a contradiction. □

Theorem 4 *No correct node is ever exposed by a correct node.*

Proof: Assume the opposite, i.e. there is a correct node i that is exposed by another correct node j. This means that ε_i contains a proof of misbehavior p, which is valid (because j, being correct, would have checked this before exposing i).

The proof p can either be `proof(inconsistent,`$a_{i,x}$`,`L`)`, or `proof(invalid,`c`,`L`)`. In the first case, $a_{i,x}$ and L must be signed by i, and L must contain an entry e_x that has the same sequence number as $a_{i,x}$, but a different hash. But i is correct, so it never uses the same sequence number twice. Thus, the second case must apply. Since p is valid, L must be signed by i, c must match the first checkpoint in L, and L must fail the conformance check for A_i. But i is correct, so it must have faithfully recorded its inputs and outputs in the log, and since A_i is deterministic by assumption, it must have produced the same outputs as the ones in L, so L cannot fail the conformance check. This is a contradiction. □