Unreliable Intrusion Detection in Distributed Computations

Dahlia Malkhi Michael Reiter

AT&T Labs—Research, Murray Hill, New Jersey, USA
{dalia,reiter}@research.att.com

Abstract

Distributed coordination is difficult, especially when the system may suffer intrusions that corrupt some component processes. In this paper we introduce the abstraction of a failure detector *that a process can use to (imperfectly) detect the corruption (Byzantine failure) of another process. In general, our failure detectors can be unreliable, both by reporting a correct process to be faulty or by reporting a faulty process to be correct. However, we show that if these detectors satisfy certain plausible properties, then the well-known distributed consensus problem can be solved. We also present a randomized protocol using failure detectors that solves the consensus problem if* either *the requisite properties of failure detectors hold* or *if certain highly probable events eventually occur. This work can be viewed as a generalization of benign failure detectors popular in the distributed computing literature.*

1 Introduction

In this paper we consider how to defend the integrity of a distributed system against an attacker that penetrates and corrupts some of its components. Primarily we want to ensure that the composite system continues to function as intended despite the presence of corrupted machines. In a stock trading application, for example, this might mean ensuring that consistent and timely data is displayed to users sitting at the other correct (uncorrupted) machines in the system, or that a transaction is consistently committed at all correct machines. Central to achieving such integrity is *distributed coordination*, and the quintessential example of a distributed coordination problem is *consensus*. Informally, in the consensus problem, each machine begins with an input, and the goal is to execute a distributed protocol by which all correct machines reach agreement on one of the values input by some machine.

In the distributed computing literature, *failure detectors* have emerged as effective tools for analyzing the behavior of distributed consensus algorithms that are tolerant of crash faults. At each step of a computation, a failure detector provides a process with a list of processes that it presently *suspects* to have crashed, allowing the process to act on this information. In general, these suspicions can be erroneous and can differ from failure suspicions at other processes, consistent with the failure detection techniques used in practice (e.g., timeouts) that failure detectors were designed to model. Chandra and Toueg showed, however, that if the failure detectors satisfy certain properties, then consensus can be solved [CT96], thereby circumventing the impossibility result of [FLP85].

In this paper we extend the failure detectors paradigm to environments in which component processes may suffer intrusions by an attacker, and we consider the consensus problem in this environment. We model an intrusion into a process as the *Byzantine failure* of that process, i.e., the process can behave arbitrarily maliciously. Intuitively, a failure detector for such an environment should detect at least some forms of Byzantine failures. However, the main difficulty in defining failure detectors for Byzantine environments is identifying which of the possible arbitrary behaviors should be detected. More specifically, failure detectors are defined according to their *Completeness*, i.e., their ability to detect actual failures, and according to their *Accuracy*, i.e., their success in avoiding false failure detection [CT96]. One might try to define "Completeness" in the Byzantine environment to require the detection of faulty processes by the other correct processes. However, this notion is ill-defined when failures are Byzantine, since, in particular, a Byzantine faulty process may visibly behave like a correct one.

The approach we take is to capture the Byzantine-faulty behavior only when it disrupts progress, and defer handling other manifestations of intrusions to the upper level consensus protocol. Thus, we de-

In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, June 1997.

fine *Completeness* to require (eventual) detection of those behaviors that may prevent progress. Intuitively, progress may be prevented when a process is waiting for a message from another process that will never arrive. Thus, we simply require the detection of those processes from which no messages will arrive. Our definition makes use of an underlying reliable broadcast primitive, which enables a process to send a message in a way that ensures that all correct processes receive the same message despite the arbitrary failure of some members (up to a third). This primitive can be implemented in our model, and thus without loss of generality, we assume that all communication within a protocol is carried via this primitive. We define a failure detector class $\diamond\mathcal{S}(bz)$ for the Byzantine environment that requires that every process from which no more broadcast messages are received is eventually detected as faulty by all the correct processes (*Strong Completeness*), and that there exists a time after which some correct process is never suspected by any of the correct processes (*Eventual Weak Accuracy*). We demonstrate that $\diamond\mathcal{S}(bz)$ suffices to solve consensus in an asynchronous, Byzantine environment, with less than a third of the system faulty.

Our analysis complements a number of previously studied failure detector classes for benign failure environments, including the weakest failure detector for solving consensus in the crash-failure environment [CHT96], the weakest conditions allowing other agreement problems to be solved in this model [GS96], and the weakest failure detector suitable for omission failure environments [DFKM96].

A limitation of designing distributed protocols using $\diamond\mathcal{S}(bz)$ is that while $\diamond\mathcal{S}(bz)$ can typically be realized in practice (e.g., using carefully tuned timeouts), there may be rare situations in which even the weak requirements of $\diamond\mathcal{S}(bz)$ cannot be guaranteed. This impossibility immediately stems from the basic impossibility result [FLP85] and the fact that a Byzantine asynchronous environment, strengthened with $\diamond\mathcal{S}(bz)$, allows solving consensus. On the other hand, a known approach to circumventing the basic impossibility result [FLP85] is to use randomization (see [R83, B83, CD89]). The drawback in randomized protocols is that there is no upper bound on the number of execution steps until a decision is made, and in theory an infinite run is possible (albeit with probability zero). A *hybrid* approach [DM94, AT96] uses both failure detection and randomization to get "the best of both worlds". Informally, a hybrid protocol ensures that if either the specified failure detection conditions hold or one of the (probable) situations created by randomizing techniques occur, then the protocol terminates correctly. This approach is promising as it enjoys the benefits of failure detectors that, in most cases, behave as specified, and "falls back" onto randomization techniques that guarantee eventual termination with probability one when bad failure scenarios occur. In this paper, we develop the first hybrid protocol suitable for a Byzantine environment, which terminates correctly if either failure detection meets the specification of $\diamond\mathcal{S}(bz)$ or if only Strong Completeness holds and a (probable) situation, influenced by randomization, finally occurs. Our approach builds on previous hybrid protocols for the crash failure environment [DM94, AT96] and on techniques that combine determinism and randomization in the synchronous environment [GP90, Z96].

2 Model

The system consists of a group of n processes, $p_0, \ldots, p_{n-1}$. Processes that follow their prescribed protocol are called *correct*, and all other processes are called *faulty*. Faulty processes may fail by crashing, or may remain alive and behave arbitrarily, i.e., incur Byzantine failures. We assume that at most $\lfloor \frac{n-1}{3} \rfloor$ processes may be faulty. Processes communicate via message passing over authenticated, reliable communication channels, guaranteeing that a message sent between two correct processes eventually reaches its destination, and ensuring that the sender of a message can be verified by the receiver. We assume that the system is asynchronous, i.e., there is no known bound on the duration of computation steps or message transfer.

2.1 Reliable broadcast

In this section we describe a communication primitive, called reliable broadcast, that processes use in our protocol. As we discuss below, this protocol can be implemented in the communication model described in the previous paragraph, and so it constitutes merely an abstraction to simplify the presentation, and not a strengthening of our basic model.

A reliable broadcast service provides the processes with two interface routines: *bcast-send*(m) for p to broadcast a message m and *bcast-receive*(m, q) for p to receive a broadcast message m from q. Reliable broadcast guarantees that all of the correct processes bcast-receive the same sequence of broadcast messages from each sender despite the malicious effort of faulty processes and even the sender. More precisely, broadcast satisfies the following properties.

Integrity: For all p and m, a correct process executes *bcast-receive*(m, q) at most once and, if q is correct, only if q executed *bcast-send*(m).

Agreement: If p and q are correct and p executes *bcast-receive*(m, r), then q executes *bcast-receive*(m, r).

Validity: If p and q are correct and p executes *bcast-send*(m), then q executes *bcast-receive*(m, p).

Source Order: If p and q are correct and both execute *bcast-receive*(m, r) and *bcast-receive*(m', r), then they do so in the same relative order and, if r is correct, in the order in which r executed *bcast-send*(m) and *bcast-send*(m').

Causal Order: If p and q are correct and p executes *bcast-receive*(m, r) before executing *bcast-send*(m'), then q executes *bcast-receive*(m, r) before executing *bcast-receive*(m', p).

Note that Agreement and Source Order together imply that for any l and any process r, the l'th bcast-receive from r is the same at all correct processes. Causal Order is weaker than causal ordering properties typically defined in the literature (e.g., [BSS91]); this weaker definition is necessitated by our consideration of Byzantine failures. There are several efficient protocols for implementing Integrity, Agreement, Validity, and Source Order in our system model (without using failure detectors or randomization); see [BT85, Rei94, MR96]. Any of these can be extended using standard techniques to implement Causal Order (e.g., [BSS91]; also see [RG95]). Our specification of reliable broadcast is thus weaker than that for consensus, which we define in Section 3.

Henceforth, we use *bcast-send/bcast-receive* as the sole means of communication among processes, and define the failure detector class $\diamond\mathcal{S}(bz)$ accordingly.

2.2 Failure detectors

Each process has access to a local failure detector module that provides it with a list of *suspected* processes. This list may change over time and can be different at different processes. If a process p is presently on process q's failure detector list, then we say that q *suspects* p.

The first step toward characterizing failure detection in the system is to identify which of the possible arbitrary failures needs to be detected. The approach we take here is to identify those failures that may prevent progress in the system and require that they be detected. For purposes of liveness detection, we assume that in any (infinite) run of the system, a correct process bcast-sends infinitely many messages. The type of failure that we then attempt to detect is one in which only finitely many messages are ever bcast-received from some process. All other failures are deferred for handling by the upper level application.

To be precise, we define *quiet* processes as follows:

Definition 1 *If in an infinite run, some correct process bcast-receives only a finite number of messages from p, then p is* quiet *in that run. Otherwise, p is* loud.

Note that in particular, correct processes are loud, and crashed processes are quiet. Faulty processes that do not participate correctly in the reliable broadcast protocol, thus preventing their own messages from being bcast-received at correct processes, are quiet. However, processes that broadcast messages not conforming to the upper level protocol are not necessarily quiet; fortunately, though, those messages could be detected by the upper level protocol.

The definition of quiet/loud processes serves us in defining the properties of failure detection as follows:

Strong Completeness: Eventually every quiet process is permanently suspected by every correct process.

Eventual Weak Accuracy: There is a time after which some correct process is not suspected by any other correct process.

The class of failure detectors defined by these two properties is called *eventually strong*, denoted $\diamond\mathcal{S}(bz)$. Note that faulty processes need not be suspected by $\diamond\mathcal{S}(bz)$, unless they are quiet. The manner in which these properties might be implemented is not of concern in this paper. We note, however, that timeouts on bcast-receives will typically provide these properties in most realistic systems.

3 Consensus using $\diamond\mathcal{S}(bz)$

In this section, we construct a protocol that uses the $\diamond\mathcal{S}(bz)$ failure detector to solve consensus in an asynchronous environment with Byzantine failures, i.e., here we assume that the system is equipped with a failure detector of class $\diamond\mathcal{S}(bz)$. This protocol demonstrates the sufficiency of $\diamond\mathcal{S}(bz)$ for solving the consensus problem in this model.

There are many variations of the consensus problem. The one we adopt in this paper is the following. Each correct process begins execution with a binary value, and a correct process completes the protocol by irrevocably *deciding* on a binary value. A consensus protocol ensures that the following properties are satisfied:

1. Every correct process decides on a value.
2. All correct processes decide on the same value, called the *consensus value*.
3. If all of the correct processes hold the same value at the beginning of the protocol, then this is the consensus value.

Our protocol for solving consensus is a variation on the Paxos consensus protocol [L89], using a revolving leader scheme and adapted for the Byzantine failure environment using techniques from [BT85]. The protocol proceeds in *rounds* that are asynchronous, i.e., rounds may overlap. In each round, a single process is the *leader*. All messages are labeled with the round for which they are intended. Throughout the protocol, each process p_i maintains a variable v_i that initially contains its input value.

Roughly speaking, a round operates as follows. It begins by each process p_i broadcasting a message containing v_i and the round-counter. When the leader has collected $\lceil\frac{2n+1}{3}\rceil$ such messages, it chooses the value that appears in $\lfloor\frac{n-1}{3}\rfloor + 1$ of the messages (and thus that was held by some correct process), and broadcasts a message suggesting that value for the consensus value. All of the processes then respond with messages containing a positive or negative acknowledgment to the suggested consensus value, where a process broadcasts a negative acknowledgment if it suspects the leader faulty. All processes collect $\lceil\frac{2n+1}{3}\rceil$ of these acknowledgment messages. If a process obtains $\lfloor\frac{n-1}{3}\rfloor + 1$ positive acknowledgments, then it changes its v_i to the suggested value. Each process then enables the next round. Moreover, if a process ever bcast-receives $\lceil\frac{2n+1}{3}\rceil$ positive acknowledgments for that round's suggested value, then it immediately decides on that value and terminates the protocol (but continues participating in the underlying broadcast protocol). Figure 1 describes the algorithm executed by p_i in pseudo-code. Initially, round number zero is enabled.

The correctness of this protocol depends critically on processes accepting only *well-formed* messages, as defined below. Throughout Figure 1, we stipulate that if a process detects a non-well-formed message from p, then it permanently adds p to its suspects list. This prevents a correct process from waiting indefinitely, e.g., in step 3, for a particular message from some other faulty process. For brevity's sake, even though all messages are broadcast to all processes, not every well-formed bcast-received message is explicitly used in the protocol by every process (except for using it to determine well-formedness; see below).

---

1. Invoke *bcast-send*($\langle A1, r, v_i\rangle$).
2. If $r \bmod n = i$ (i.e., I'm the leader), and well-formed messages $\langle A1, r, *\rangle$ are bcast-received from $\lceil\frac{2n+1}{3}\rceil$ processes, then let v be the value that appears in at least $\lfloor\frac{n-1}{3}\rfloor + 1$ of the messages. Execute *bcast-send*($\langle L1, r, v\rangle$).
3. Wait until either a well-formed message $\langle L1, r, v\rangle$ is bcast-received from the leader or until the leader is suspected faulty, whichever occurs first. In the former case, set v_i to v and execute *bcast-send*($\langle A2, r, v, ACK\rangle$). In the latter case, execute *bcast-send*($\langle A2, r, \perp, NACK\rangle$).
4. Once well-formed messages $\langle A2, r, *, ACK/NACK\rangle$ have been bcast-received from $\lceil\frac{2n+1}{3}\rceil$ different processes, if $\lfloor\frac{n-1}{3}\rfloor + 1$ of these messages are of the form $\langle A2, r, v, ACK\rangle$ for a common v, set $v_i = v$. In any case, enable round $r + 1$.
5. If ever well-formed messages $\langle A2, r, v, ACK\rangle$ are bcast-received from $\lceil\frac{2n+1}{3}\rceil$ processes for a common value v, then decide v and terminate the protocol.

Figure 1. Consensus protocol using $\Diamond\mathcal{S}(bz)$; Round r at process p_i

---

A well-formed message is one that is neither *malformed*, *out-of-order*, or *unjustifiable*, defined as follows:

1. A *malformed message* is one that is internally inconsistent with the protocol specification, i.e., a message that the sender would never generate in any run in which it were correct.
2. An *out-of-order message* is a message that the sender sends either too early or too late in its sequence of messages. More precisely, our protocol prescribes an exact sequence of message types to be sent per round for each process, and this sequence should never be intertwined with those the process sends in another round. So, if a process p bcast-receives from q either (i) round r messages out of sequence or multiple times, (ii) messages for round r after bcast-receiving messages from q for some round $r' > r$, or (iii) messages for round r before bcast-receiving all prescribed messages from q for all rounds $r' < r$, then p has detected out-of-order messages from q. Detecting out-of-order messages is essential to en-

suring progress, because it prevents a faulty process from sending infinitely many messages in a single round or skipping a round altogether, leaving other processes waiting on its messages.

3. An *unjustifiable message* is one that its sender, if correct, could not possibly have sent based upon the messages that it bcast-received prior to sending it. That is, by Causal Order, a process p that executes *bcast-receive*(m, q) has also bcast-received every message that q had bcast-received when q executed *bcast-send*(m) (provided that q is correct). If it is impossible that q correctly constructed m based on what it previously bcast-received—i.e., m is inconsistent with the messages that p bcast-received by the time p executes *bcast-receive*(m, q)—then p has detected an unjustifiable message from q. In particular, the $\langle L1, r, v\rangle$ message in Figure 1 is justifiable only if v appears in $\lfloor\frac{n-1}{3}\rfloor + 1$ well-formed $A1$ messages by the time it is bcast-received. To make p's determination of whether m is justifiable more efficient, q could include in m explicit references to messages that justify m.

A proof of correctness for this protocol involves an intricate induction on round numbers and steps. Below, we provide an intuitive sketch of the proof.

Theorem 1 *Any two correct processes that decide, decide on the same value.*

Proof. (Sketch) Suppose that p decides on v in round r, q decides v' in round r', and, without loss of generality, that $r < r'$ (the case $r = r'$ is trivially satisfied). For p to decide v in round r, p must have bcast-received $\lceil\frac{2n+1}{3}\rceil$ messages of the form $\langle A2, r, v, ACK\rangle$. This implies that every correct process p_j bcast-delivers at least $\lfloor\frac{n-1}{3}\rfloor + 1$ $\langle A2, r, v, ACK\rangle$ (and no more than $\lfloor\frac{n-1}{3}\rfloor$ $A2$ messages with value other than v), and thus, $v_j = v$ when it enables round $r + 1$ and. Using a simple induction, this hold when p_j enables round $r' > r$. Thus the well-formed *L1* message sent by the leader of round r' must be of the form $\langle L1, r', v, A\rangle$ for some A, and the result follows. □

Theorem 2 *Eventually all correct processes decide.*

Proof. (Sketch) First note that if round r is enabled at p, where p is correct, and p does not decide in a round $r' \le r$, then round $r+1$ is eventually enabled at p. Second, note that if any correct process decides at some round r, then eventually the bcast-receives that caused it to decide will cause all of the other correct processes to decide as well. Let t be some time after which some correct process p is never suspected by any other correct process; t is guaranteed to exist by Eventual Weak Accuracy. Assume that no correct process decides until some time $t' > t$, at which round r is enabled at all correct processes and for which p is the leader. By assumption, since p did not decide before round r, p sends a (well-formed) *L1* message in round r and all correct processes reply with an ACK message, thus causing all correct processes to decide. □

Theorem 3 *If all the correct processes hold the same value v at the beginning of the protocol, then v is the consensus value.*

Proof. (Sketch) If all correct processes hold the same value v at the beginning of the protocol, then an induction similar to Theorem 1's shows that $v_j = v$ at every correct process p_j when any round is enabled. Thus, v is the only value that could be the consensus value. □

4 A hybrid protocol

In this section, we present a hybrid consensus protocol that uses both a failure detector and randomization techniques. The protocol is guaranteed to terminate if either the failure detector satisfies the requirements of $\Diamond\mathcal{S}(bz)$, or if only Strong Completeness holds and certain probable scenarios, which are influenced by randomization, finally occur. In other words, if the processes have access to a failure detector in class $\Diamond\mathcal{S}(bz)$ then the protocol is guaranteed to terminate, and in addition, if the processes have access to a source of random bits then the protocol is guaranteed to terminate with probability 1, even if Eventual Weak Accuracy does not hold (Strong Completeness is typically easy to satisfy, e.g., using timeouts).

We begin with a high level description of the protocol, which is a variation on the protocol above (Figure 1). As above, the protocol operates in logical rounds, each having one designated process as leader; the leader revolves among all of the processes over the rounds. To distinguish messages in the protocol below from the ones in Figure 1, we denote their type field with an overhead bar, as in $\overline{A1}$.

Each process p_i again begins round r by broadcasting a message containing v_i and the round counter. As we will see below, to be justifiable, v_i must appear in $\lfloor\frac{n-1}{3}\rfloor + 1$ of certain messages bcast-received in round $r-1$. Once p_i collects $\lceil\frac{2n+1}{3}\rceil$ of these messages, if all of the messages contain the same value v (they will if any correct process decided v in a round $r' < r$), then p_i sets v_i to v; otherwise v_i is set to $\perp$. If the leader p_j now has $v_j = \perp$, it sets v_j to a random value. In any case, it broadcasts a message containing v_j. Each process

p_i waits until it either bcast-receives this message from the leader, in which case it sets v_i to the leader's value, or until it suspects the leader, in which case it sets v_i to a random value if $v_i = \perp$. Process p_i then broadcasts a message containing v_i, and waits to collect $\lceil\frac{2n+1}{3}\rceil$ of these messages from other processes. Among these messages, there will be at least $\lfloor\frac{n-1}{3}\rfloor + 1$ containing a value v; p_i sets v_i to v and enables the next round. Finally, if a process ever bcast-receives $\lceil\frac{2n+1}{3}\rceil$ of these messages containing the value v, it decides on this value and terminates the protocol (but continues participating in the underlying broadcast protocol). Figure 2 below contains a pseudo-code description of the hybrid consensus protocol executed at process p_i. Initially, round number zero is enabled.

---

1. Invoke *bcast-send*($\langle\overline{A1}, r, v_i\rangle$).
2. Wait until well-formed messages $\langle\overline{A1}, r, v_j\rangle$ have been bcast-received from a set P of $\lceil\frac{2n+1}{3}\rceil$ processes. Then set
$$v_i = \begin{cases} v & \forall p_j \in P : v_j = v \\ \text{random value} & \text{otherwise, and } r \bmod n = i \\ \perp & \text{otherwise} \end{cases}$$
3. If $r \bmod n = i$ (i.e., I'm the leader), perform *bcast-send*($\langle\overline{L1},\ r,\ v_i\rangle$).
4. Wait until a well-formed message $\langle\overline{L1},\ r,\ v\rangle$ is bcast-received from the leader, or until the leader is suspected faulty, whichever occurs first. In the former case, set v_i to v. Otherwise, if $v_i = \perp$, then set v_i to a randomly drawn value. Finally, execute *bcast-send*($\langle\overline{A2},\ r,\ v_i\rangle$).
5. Once well-formed messages $\langle\overline{A2},\ r,\ *\rangle$ have been bcast-received from $\lceil\frac{2n+1}{3}\rceil$ processes, set v_i equal to the value v that occurs in at least $\lfloor\frac{n-1}{3}\rfloor + 1$ of these messages. Enable round $r + 1$.
6. If ever well-formed messages $\langle\overline{A2},\ r,\ v\rangle$ are bcast-received from $\lceil\frac{2n+1}{3}\rceil$ processes for a common value v, then decide v and terminate this protocol.

Figure 2. Hybrid consensus protocol using $\diamond\mathcal{S}(bz)$; Round r at p_i

---

As in the protocol of the previous section, the processes in this protocol look for malformed, out-of-order, or unjustifiable messages. In this protocol, a message $\langle\overline{A1}, r, 0\rangle$ is justifiable for $r > 0$ only if $\lfloor\frac{n-1}{3}\rfloor + 1$ well-formed messages of the form $\langle\overline{A2}, r-1, 0\rangle$ were bcast-received in the previous round (and similarly for $\langle\overline{A1}, r, 1\rangle$). Round 0's $\overline{A1}$ messages are justified similarly by $\overline{A0}$ messages exchanged during a one-time preparatory step, executed at the start of the protocol, as depicted in Figure 3. A message $\langle\overline{L1}, r, 0\rangle$ or $\langle\overline{A2}, r, 0\rangle$ is justifiable only if some well-formed $\overline{A1}$ message for round r of the form $\langle\overline{A1}, r, 0\rangle$ was bcast-received (and similarly for $\langle\overline{L1}, r, 1\rangle$ and $\langle\overline{A2}, r, 1\rangle$).

0a. Invoke *bcast-send*($\langle\overline{A0}, 0, v_i\rangle$).

0b. Wait until well-formed messages $\langle\overline{A0}, 0, *\rangle$ have been bcast-received from $\lceil\frac{2n+1}{3}\rceil$ processes, and set v_i to the value that appears in at least $\lfloor\frac{n-1}{3}\rfloor + 1$ of them.

Figure 3. Preparatory step for round 0 at process p_i

Theorem 4 *Let p and q be two correct processes that decide on v, and v', respectively. Then $v = v'$.*

Proof. (Sketch) Let p decide in round r, q decide in r', and without loss of generality, assume $r' > r$ (the case $r' = r$ is trivially satisfied). For p to decide on the value v in some round r, p must bcast-receive $\lceil\frac{2n+1}{3}\rceil$ well-formed messages of the form $\langle\overline{A2}, r, v\rangle$. Therefore, in round $r+1$, and by simple induction, any $r' > r$, every well-formed $\overline{A1}$ message contains the value v. Therefore, the only possible consensus value from round r and on is v, and thus $v' = v$. $\square$

Lemma 1 *For any round r in which all the correct processes participate and the leader is correct, there is a positive probability that all the correct processes broadcast identical $\overline{A2}$ messages (causing a decision on that value to eventually occur).*

Proof. (Sketch) Let p_i and p_j be any two correct processes that participate in round r. If at the end of step 2, $v_i \neq \perp$ and $v_j \neq \perp$, and neither v_i nor v_j was chosen at random in step 2, then $v_i = v_j$; let v denote this common value (if it exists). If the leader p_k of round r chooses v_k at random in step 2, then it has a positive probability of setting $v_k = v$. Moreover, every other correct process p_i that has $v_i = \perp$ at the end of step 2 and that suspects the leader faulty has a positive probability of setting $v_i = v$ in step 4. Therefore, there is a positive probability that all correct processes send the same $\overline{A2}$ messages in step 4. $\square$

Lemma 2 *Let t be some time after which some correct process p_i is never suspected by any other correct process. Let r be a round that is enabled after time t, for which p_i is the leader, and in which all the correct processes participate. Then all the correct processes decide at round r (at the latest).*

Proof. (Sketch) By assumption, all the correct processes bcast-receive a well-formed $\overline{L1}$ message from p_i in round r and broadcast well-formed $\overline{A2}$ messages containing the same value. Eventually, the bcast-receives of these messages will cause every correct process to decide on that value. □

Theorem 5 *Assume that either all correct processes are equipped with failure detectors that satisfy the properties of $\diamond\mathcal{S}(bz)$, or all correct processes are equipped with unbiased random bit generators and failure detectors that satisfy Strong Completeness. Then in any execution of the protocol above, eventually all correct processes decide (with probability 1).*

Proof. (Sketch) First, note that if any correct process decides at round r, then eventually the bcast-receives that caused the decision will cause all of the correct processes to decide as well. Second, note that every correct process executing round r eventually either decides or enables round $r+1$. The theorem now follows immediately from Lemmata 1 and 2. □

Theorem 6 *If all the correct processes hold the same value v at the beginning of the protocol, then v is the consensus value.*

Proof. (Sketch) If all correct processes hold the same value v at the beginning of the protocol, then an induction similar to Theorem 4's shows that $v_j = v$ at every correct process p_j when any round is enabled, and thus, every well-formed $\overline{A1}$ message contains v. Therefore, v is the only value that could be the consensus value. □

5 Optimizations

The protocols presented above were designed to demonstrate solvability, without taking into account performance considerations. These protocols, and in particular the hybrid consensus protocol, can be optimized in several ways.

We begin by considering the behavior of the hybrid protocol in the (common) case where the system is predictable. For example, in many practical situations, a failure detector can be tuned so as to make very few mistakes, and it is therefore desirable to have a protocol that terminates quickly when each correct process' failure detector is *ideal*—i.e., when a process is suspected if and only if no more messages will ever be bcast-received from that process. In the protocol above, however, it may take several rounds until a round whose leader is correct executes (and then it terminates), even when the failure detector behaves ideally. To achieve early termination in this case, we replace step 1 of the hybrid protocol with the following:

1a. Invoke *bcast-send*($\langle\overline{A1'}, r, v_i\rangle$).

1b. Wait until well-formed messages $\langle\overline{A1'}, r, v_j\rangle$ have been bcast-received from a set P of processes such that $|P| \geq \lceil\frac{2n+1}{3}\rceil$, and all processes not in P are suspected. Then set $v_i = 0$ if at least $\lfloor\frac{n-1}{3}\rfloor + 1$ of these messages contain 0, and $v_i = 1$ otherwise.

1c. Invoke *bcast-send*($\langle\overline{A1}, r, v_i\rangle$).

Briefly, using the optimization above, an $\overline{A1'}$ message is justifiable as $\overline{A1}$ before, whereas a message $\langle\overline{A1}, r, 0\rangle$ is now justifiable if $\lfloor\frac{n-1}{3}\rfloor + 1$ well-formed messages of the form $\langle\overline{A1'}, r, 0\rangle$ were bcast-received in round r (and similarly for $\langle\overline{A1}, r, 1\rangle$).

It is easy to verify that these steps do not violate any of the correctness claims made for the protocol. In addition, the modified protocol maintains:

Theorem 7 *Let t be a time after which all correct processes' failure detectors are ideal. Then the modified protocol terminates (at the latest) at the end of the first asynchronous round following t.*

Aguilera and Toueg [AT96] describe another common situation, where very few or no failures occur, the leader of the first round is correct, and few enough (or no) false failure detections are made so that the first leader is not suspected by any correct process. In such a case, the hybrid protocol above terminates after one round (with four phases of message-exchanges).

Finally, we consider the behavior of the protocol in extreme situations in which the $\diamond\mathcal{S}(bz)$ failure detector properties are not satisfied. In this case, the randomizing techniques we apply assure that the probability that the protocol terminates approaches unity as more rounds are executed. However, the expected number of rounds until termination in the hybrid protocol above is fairly high ($O(n2^n)$). This can be improved using techniques for *distributed coin tossing*. To employ such methods, a process replaces the random bit drawing of steps 2 and 4 of the hybrid protocol with calls to a *coin-toss* procedure, returning a binary value that is expected to become identical at all of the correct processes within some known number of invocations. Ben-Or gave a method for tossing a coin in an asynchronous environment with up to $t < \sqrt{n}$ Byzantine faulty processes [B83], that is expected to produce such a unanimous coin toss within a constant number of invocations. He later improved this with another protocol that is resilient to up to $t < \frac{n}{7}$ Byzantine failures [B85]. Canetti and Rabin [CR93] used an asynchronous verifiable secret sharing scheme to achieve a coin-tossing protocol resilient to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine failures that has constant time expected termination with overwhelming probability. Another technique to speed up termination is to use precomputed random bits at runtime. Methods to distribute shares of secret bits using a trusted dealer have been suggested by Rabin [R83] and Toueg [T84].

6 Conclusions

In this paper, we have shown how to build practical consensus protocols for environments that may suffer intrusions by an attacker. Since consensus underlies a large class of distributed coordination problems, our protocols provide insight into a practical approach to achieve distributed coordination despite the participation of corrupted machines. We demonstrated that unreliable failure detectors can be used to achieve consensus in such environments, provided that certain (weak) constraints hold. In most reasonable practical settings, the deterministic protocol we presented terminates fairly quickly, i.e., as soon as the system is stable enough for one correct process to be unsuspected by all of the other correct processes (Eventual Weak Accuracy). To cope with scenarios in which even these weak constraints fail, we incorporated randomization techniques and produced a hybrid protocol that is guaranteed to succeed with probability one even when failure detection is continually erroneous.

Our protocols were designed to cope with up to $\lfloor \frac{n-1}{3} \rfloor$ Byzantine failures of participating servers. In practice, other failure assumptions may hold, including less uniform ones. In the future, we plan to extend the treatment here to deal with other Byzantine failure assumptions by employing *Byzantine quorum systems* [MR97] to derive the consistency and liveness of our protocols, thus replacing the uniform requirement thresholds of $\lceil \frac{2n+1}{3} \rceil$ and $\lfloor \frac{n-1}{3} \rfloor + 1$ in our protocols.

An open problem left by this work is characterizing the weakest conditions (equivalently, the weakest failure detector) allowing consensus to be solved in Byzantine environments.

References

[AT96] M. K Aguilera and S. Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG)*, LNCS 1151, pp. 29–39, October 1996.

[B83] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pp. 27–30, 1983.

[B85] M. Ben-Or. Fast asynchronous Byzantine agreement. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, pp. 149-151, 1985.

[BSS91] K. P. Birman, A. Schiper and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM* 32(4):824–840, October 1985.

[CR93] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pp. 42–51, 1993.

[CD89] B. Chor and C. Dwork. Randomization in Byzantine agreement. *In Advances in Computing Research, Randomness in Computation*, volume 5, JAI Press, edited by S. Micali, pp. 443–497, 1989.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CHT96] T. D. Chandra, V. Hadzilacos and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[DFKM96] D. Dolev, R. Friedman, I. Keidar and D. Malkhi. Failure detectors in omission failure environments. Technical Report 96-1605, Department of Computer Science, Cornell University, September 1996.

[DM94] D. Dolev and D. Malki. Consensus made practical. Technical Report 94-7, Institute of Computer Science, the Hebrew University of Jerusalem, March 1994.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2):374–382, April 1985.

[GP90] O. Goldreich and E. Petrank. The best of both worlds: Guaranteeing termination in fast Byzantine agreement protocols. *Information Processing Letters*, Vol. 36, pp. 45–49, October 1990.

[GS96] R. Guerraoui and A. Schiper. "Γ-Accurate" failure detectors. *In the 10th International Workshop on Distributed Algorithms (WDAG)*, LNCS 1151, pp. 269–286, October 1996.

[L89] L. Lamport. The part-time parliament. *Systems Research Center, Digital Equipment Corp., Palo Alto*, Technical Report 49, September 1989.

[MR96] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. *In Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pp. 9–17, June 1996. Also, to appear in *Journal of Computer Security*.

[MR97] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, May 1997. To appear.

[R83] M. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th annual ACM Symposium on Foundations of Computer Science*, pp. 403–409, 1983.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pp. 68–80, November 1994.

[RG95] M. K. Reiter and L. Gong. Securing causal relationships in distributed systems. *The Computer Journal* 38(8):633–642, Oxford University Press, 1995.

[T84] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pp. 163-178, 1984.

[Z96] A. Zamsky. A randomized Byzantine agreement protocol with constant expected time and guaranteed termination in optimal (deterministic) time. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pp. 201–208, 1996.