

Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector*

Kim Potter Kihlstrom, L. E. Moser, P. M. Melliar-Smith

Department of Electrical and Computer Engineering

University of California, Santa Barbara, CA 93106

kimk@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

Unreliable fault detectors can be used to solve the consensus problem in asynchronous distributed systems that are subject to crash faults. We extend this result to asynchronous distributed systems that are subject to Byzantine faults. We define the class $\diamond\mathcal{S}(\text{Byz})$ of eventually strong Byzantine fault detectors and the class $\diamond\mathcal{W}(\text{Byz})$ of eventually weak Byzantine fault detectors and show that any Byzantine fault detector in $\diamond\mathcal{W}(\text{Byz})$ can be transformed into a Byzantine fault detector in $\diamond\mathcal{S}(\text{Byz})$. We present an algorithm that uses a fault detector in $\diamond\mathcal{S}(\text{Byz})$ to solve the consensus problem in an asynchronous distributed system with at most $\lfloor (n-1)/3 \rfloor$ Byzantine faults. The class $\diamond\mathcal{W}(\text{Byz})$ of Byzantine fault detectors is the weakest class of fault detectors that can be used to solve consensus in such an asynchronous distributed system.

Keywords: Consensus, asynchronous systems, Byzantine fault, unreliable fault (failure) detector, distributed algorithms

1 Introduction

Consensus is a fundamental problem in distributed computing. Fischer, Lynch and Paterson [4] have shown that it is impossible to achieve consensus in an asynchronous distributed system that is subject to even one crash fault. Chandra and Toueg [2] have shown, however, that consensus is possible in an asynchronous system that is subject to crash faults if an unreliable fault detector¹ is provided. The fault detector is unreliable in that it may suspect a correct process or it may fail to suspect a faulty process.

*Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under grant number F30602-95-1-0048. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency and Rome Laboratory or the U.S. Government.

¹Popular terminology refers to these as unreliable failure detectors but, according to the ISO standard definitions of fault and failure, they should more properly be called unreliable fault detectors.

In [2] Chandra and Toueg have identified two critical properties of unreliable fault detectors: completeness and accuracy. Their $\diamond\mathcal{S}$ class of unreliable fault detectors satisfies eventual strong completeness and eventual weak accuracy, whereas their $\diamond\mathcal{W}$ class of unreliable fault detectors satisfies eventual weak completeness and eventual weak accuracy. Chandra and Toueg have given an algorithm that transforms any fault detector in the class $\diamond\mathcal{W}$ into a fault detector in the class $\diamond\mathcal{S}$. The transformation algorithm strengthens eventual weak completeness while preserving eventual weak accuracy. They have also given an algorithm that uses any fault detector in $\diamond\mathcal{S}$ to solve consensus in an asynchronous distributed system that is subject to at most $\lfloor (n-1)/2 \rfloor$ crash faults. In [1] Chandra, Hadzilacos and Toueg have proved that $\diamond\mathcal{W}$ is the class of weakest fault detectors that can be used to solve consensus in such a system.

In this paper we extend the work of Chandra and Toueg and of Chandra, Hadzilacos and Toueg on unreliable fault detectors for crash faults, by considering unreliable fault detectors for Byzantine faults. We define two classes of unreliable Byzantine fault detectors: the class $\diamond\mathcal{S}(\text{Byz})$ of eventually strong Byzantine fault detectors and the class $\diamond\mathcal{W}(\text{Byz})$ of eventually weak Byzantine fault detectors. Next we show that any Byzantine fault detector in $\diamond\mathcal{W}(\text{Byz})$ can be transformed into a Byzantine fault detector in $\diamond\mathcal{S}(\text{Byz})$. Then we present an algorithm that uses a fault detector in $\diamond\mathcal{S}(\text{Byz})$ to solve the consensus problem in an asynchronous distributed system with at most $\lfloor (n-1)/3 \rfloor$ Byzantine faults. A fault detector in $\diamond\mathcal{W}(\text{Byz})$ can, therefore, also be used to solve consensus in such a system. Next we present an implementation of a Byzantine fault detector that can be used with the consensus algorithm. The fault detectors in $\diamond\mathcal{W}(\text{Byz})$ are the weakest fault detectors for solving consensus in an asynchronous distributed system subject to Byzantine faults.

In [10] Malkhi and Reiter defined a class $\diamond\mathcal{S}(\text{bz})$ of eventually strong fault detectors and showed that a fault detector in $\diamond\mathcal{S}(\text{bz})$ can be used to solve consensus in an asynchronous distributed system that is subject to at most $\lfloor (n-1)/3 \rfloor$ Byzantine faults. All messages in their consensus algorithm are sent using a source-ordered, causally-ordered reliable broadcast service. We do not use that approach because the algorithm used to implement the reliable broadcast service is very expensive. Malkhi and Reiter do not define a class of eventually weak fault detectors.

2 The Model

We consider a distributed system consisting of n processes where $n \geq 2$; each process in the system has a unique process identifier. The system is asynchronous in that no bound can be placed on the time required for a computation or for communication of a message. Processes have access to local clocks, but the clocks are not synchronized. The communication network consists of communication channels on which pairs of processes communicate via messages using the *send* and *receive* primitives; the communication network is assumed not to partition. The communication channels are reliable in that messages that are sent between correct processes are eventually received and are not modified by the communication medium.

Processes may be either correct or faulty. Correct processes always behave according to their specification. Faulty processes exhibit arbitrary (Byzantine)

behavior. Because a Byzantine process may behave as if it crashed, we include crash faults among the Byzantine faults we consider. We let k denote the maximum number of Byzantine processes, and require that $k \leq \lfloor (n-1)/3 \rfloor$. Thus, at least $\lceil (2n+1)/3 \rceil$ processes are correct.

We employ a public key cryptosystem, such as RSA [12], in which each process p possesses a private key known only to itself with which it can digitally sign messages. We refer to a message $\langle - \rangle$ signed by p as $\langle - \rangle_p$.² Each process is able to obtain the public keys of other processes to verify the identity of the sender of a message. We assume that a signature is unforgeable, and that a key distribution service is available such that all correct processes have the same public key for each of the processes. Thus, our fault model is a Byzantine fault model with authentication.

In the consensus problem, each process begins with an input value and all of the correct processes must decide on a common output value, called the decision value, that is related to the input values. There are several different variations of the consensus problem. The consensus problem considered here is the general consensus problem for which the input values and the decision value need not be binary. A solution to the consensus problem is an algorithm that satisfies the following properties:

Termination. Every correct process eventually decides on some value.

Validity. If all correct processes have the same input value, then any correct process that decides must decide on that value.

Agreement. No two correct processes decide differently.

Irrevocability. Once a correct process decides on a value, it remains decided on that value.

The validity property only restricts the decision value in the case in which all correct processes hold the same input value and is therefore weaker than the validity property that might be desired. In particular, if all correct processes do not hold the same input value, the decision value can be a value proposed by a Byzantine process; in such a situation it may not even be the input value of the Byzantine process. It is possible to strengthen the validity property in our environment if the input values and decision value are restricted to binary values; this is discussed further in Section 4.

3 Unreliable Byzantine Fault Detectors

We distinguish between *detectable* and *non-detectable* Byzantine faults. We characterize these types of faults in Section 6, where we describe a fault detector that detects detectable Byzantine faults.

We define two classes of unreliable Byzantine fault detectors: the class $\diamond S(\text{Byz})$ and the class $\diamond W(\text{Byz})$. A fault detector is in $\diamond S(\text{Byz})$ if it satisfies the following properties:

Eventual Strong Byzantine Completeness. There is a time after which every process that has exhibited a detectable Byzantine fault is permanently suspected by every correct process.

²Throughout this paper, we use $-$ to refer to an arbitrary value of an appropriate type.

Eventual Weak Accuracy. There is a time after which some correct process is never suspected by any correct process.

We could attempt to define an eventual weak Byzantine completeness property as follows: There is a time after which every process that has exhibited a detectable Byzantine fault is permanently suspected by some correct process. However, in a system that is subject to Byzantine faults, a transformation algorithm similar to the transformation algorithm of [2], although it transforms the eventual weak Byzantine completeness property above into eventual strong Byzantine completeness, cannot ensure that eventual weak accuracy is preserved in the presence of even a single Byzantine fault. In the algorithm of [2], each process periodically sends the suspects list obtained from its local fault detector to every process. When a process p receives a suspects list from another process q , it merges that list with its own suspects list, and removes process q if q appears in its list. The problem with such an algorithm in a Byzantine environment is that a Byzantine process can report repeatedly that it suspects every process, causing the fault detector obtained from the transformation to report repeatedly that every process is faulty. Thus, even though the eventually weak fault detector at a correct process satisfies eventual weak accuracy, the transformed fault detector does not.

To address this difficulty, we consider an alternative transformation algorithm in which a process p reports another process q to be faulty only if p has received reports from $k + 1$ processes (possibly including itself) whose eventually weak fault detectors regard q as faulty. While such an algorithm preserves eventual weak accuracy, it does not transform the eventual weak Byzantine completeness property given above into eventual strong Byzantine completeness. The problem is that the property given above requires only that there exists *one* correct process that suspects the faulty process and does not guarantee the $k + 1$ reports required. We therefore define a new eventual weak Byzantine completeness property as follows:

Eventual Weak Byzantine $(k + 1)$ -Completeness. There is a time after which every process that has exhibited a detectable Byzantine fault is permanently suspected by at least $k + 1$ correct processes.

Then we define $\diamond\mathcal{W}(\text{Byz})$ to be the class of unreliable fault detectors that satisfy eventual weak Byzantine $(k + 1)$ -completeness and eventual weak accuracy.

A fault detector \mathcal{D} in $\diamond\mathcal{W}(\text{Byz})$ can be transformed into a fault detector \mathcal{D}' in $\diamond\mathcal{S}(\text{Byz})$ using the transformation algorithm given in Figure 1. This algorithm makes use of techniques from the transformation algorithm of [5]. It uses \mathcal{D} to maintain a variable $output_p$ at each process p . This variable emulates the output of \mathcal{D}' at process p .

Theorem 1. The algorithm given in Figure 1 transforms eventual weak Byzantine $(k + 1)$ -completeness into eventual strong Byzantine completeness while preserving eventual weak accuracy in an asynchronous distributed system.

The proof of Theorem 1 can be found in the full version of the paper [6].

A fault detector \mathcal{D}' in $\diamond\mathcal{S}(\text{Byz})$ can be transformed into a fault detector \mathcal{D} in $\diamond\mathcal{W}(\text{Byz})$ by a trivial reduction algorithm in which each process periodically writes

```

/* Each process p executes the following */
/* Initialization */
outputp ← ∅;                                     /* outputp emulates D'p */
suspectedp ← ∅;                                   /* The set of processes suspected by p */
for each r in S
    suspectingp[r] ← ∅;                             /* The set of processes p thinks are currently suspecting r */
initiate concurrent tasks 1 and 2;

/* Task 1: */
repeat forever
    suspectedp ← Dp;                               /* p queries its local fault detector Dp */
    send ⟨p, suspectedp⟩p to all;

/* Task 2: */
when receive ⟨q, suspectedq⟩q from q
    for each r in S
        if r ∈ suspectedq then
            suspectingp[r] ← (suspectingp[r] ∪ {q});
        else
            suspectingp[r] ← (suspectingp[r] - {q});
        if |suspectingp[r]| ≥ k + 1 then
            outputp ← (outputp ∪ {r});
        else
            outputp ← (outputp - {r});

```

Figure 1: Algorithm to transform a fault detector $\mathcal{D} \in \diamond\mathcal{W}(\text{Byz})$ into a fault detector $\mathcal{D}' \in \diamond\mathcal{S}(\text{Byz})$.

the suspects list obtained from its local fault detector into $output_p$ [2]. Thus, the classes $\diamond\mathcal{S}(\text{Byz})$ and $\diamond\mathcal{W}(\text{Byz})$ are equivalent.

4 Solving Consensus

We now present an algorithm that solves the consensus problem in an asynchronous distributed system that is subject to Byzantine faults, provided that at most $\lfloor (n-1)/3 \rfloor$ processes are faulty, using a fault detector in $\diamond\mathcal{S}(\text{Byz})$. Pseudocode for the consensus algorithm is given in Figure 2.

The consensus algorithm, which uses techniques from the consensus algorithm of [2], employs a rotating coordinator and proceeds in asynchronous rounds. All processes have *a priori* knowledge that during round r , the coordinator is process $c \equiv (r \bmod n) + 1$, where n is the number of processes in the system.

Each process p maintains several local variables including its estimate e_p of the decision value, its current round number r_p , the current coordinator c_p , and the timestamp ts_p . The timestamp is the number of the latest round in which p updated e_p , and is initially equal to 0. The algorithm uses four types of messages, *estimate*, *select*, *confirm* and *ready*, which are described below.

The algorithm consists of three concurrent tasks. The first task (Selection) consists of a sequence of rounds, each round of which contains three phases. In the first phase, each process sends an *estimate* message that contains its estimate e_p to the coordinator.

In the second phase, the coordinator waits to receive *estimate* messages from $\lfloor (2n+1)/3 \rfloor$ processes, chooses a value es based on the *estimate* messages it has

received, and sends a *select* message to all. Because there are at most $\lfloor (n-1)/3 \rfloor$ faulty processes, the coordinator will receive *estimate* messages from at least $\lceil (2n+1)/3 \rceil$ processes, and thus will not wait forever in the second phase.

In the third phase, each process waits to receive *confirm* messages with the same value for the round³ from $\lceil (2n+1)/3 \rceil$ distinct processes or until it suspects the coordinator. If a process receives *confirm* messages containing the same value from $\lceil (2n+1)/3 \rceil$ distinct processes, it updates its value e_p , its timestamp ts_p and its set of *confirm* messages, and sends a *ready* message to all. If the coordinator is faulty, then some correct process p may not receive *confirm* messages containing the same value from $\lceil (2n+1)/3 \rceil$ distinct processes. In this case, by the eventual strong accuracy property of the fault detector, p will eventually come to suspect the coordinator. Thus, no correct process will wait forever in the third phase.

In the second task (Confirmation), whenever a process receives a *select* message for any round from the coordinator of that round, and it has not already sent a *confirm* message for that round, it sends a *confirm* message to all. The *confirm* message contains the value e found in the *select* message.

In the third task (Decision), a process waits to receive *ready* messages containing a common value for the same round from $\lceil (2n+1)/3 \rceil$ distinct processes, and then decides on that value. The third task then terminates, although execution of the first and second tasks continues. It is possible to terminate the first two tasks also, but for simplicity of presentation we choose not to do so here.

Once there is a round in which a correct process can decide on a value v , then the value v is locked, using the timestamp, so that no other correct process can decide on a different value. For a correct process to decide on a value v , it must have received *ready* messages containing the value v and a common round r from $\lceil (2n+1)/3 \rceil$ distinct processes. At least $\lfloor (n-1)/3 \rfloor + 1$ of those *ready* messages were sent by correct processes for round r . In any round $r' > r$, each of those correct processes will send an *estimate* message, containing the value v and timestamp $ts_p \geq r$, to the coordinator. Because the coordinator must wait for *estimate* messages from at least $\lceil (2n+1)/3 \rceil$ processes, the coordinator will wait for an *estimate* message from at least one of those correct processes. The coordinator must select the value from one of those *estimate* messages with the highest timestamp and, thus, will select the value v .

If the largest timestamp ts of the $\lceil (2n+1)/3 \rceil$ *estimate* messages that the coordinator received is the value 0, and if at least $\lfloor (n-1)/3 \rfloor + 1$ of those *estimate* messages contain a common value e , the coordinator chooses the value e in its *select* message. This step is necessary to satisfy the validity property. If a stronger validity property is required, it is possible to solve the binary consensus problem by modifying this step of the algorithm such that, if $ts = 0$, the coordinator always chooses the value contained in at least $\lfloor (n-1)/3 \rfloor + 1$ of the *estimate* messages it received. Such an algorithm would satisfy the following validity property: If a correct process decides on a value v , then v is the initial value of a correct process.

To prevent one process from masquerading as another, messages are signed; a process that receives a message must verify the signature on the message. If the

³The statement “a process p sends a *confirm* message for round r ” does not imply that p is executing round r in task 1 at the time it sends that *confirm* message in task 2 and, similarly, “ p decides for round r .”

```

/* Each process p executes the following */
/* Initialization */
 $e_p \leftarrow v_p$ ;          /*  $v_p$  is the initial value held by p;  $e_p$  is p's estimate of the decision value */
 $r_p \leftarrow 0$ ;          /*  $r_p$  is p's current round number */
 $ts_p \leftarrow 0$ ;        /*  $ts_p$  is the last round in which p updated  $e_p$  */
 $confirm_{s_p} \leftarrow \emptyset$ ; /*  $confirm_{s_p}$  is the set of confirm messages that caused p to update  $e_p$  */
initiate concurrent tasks 1, 2 and 3;

/* Task 1: Selection */
repeat forever
   $r_p \leftarrow r_p + 1$ ;          /* Rotate through coordinators */
   $c_p \leftarrow (r_p \bmod n) + 1$ ; /*  $c_p$  is the current coordinator */

  /* Phase 1: */
  send  $\langle (estimate, p, r_p, e_p, ts_p)_p, confirm_{s_p} \rangle_p$  to  $c_p$ ;

  /* Phase 2: */
  if [ $p = c_p$ ] then
    wait until [for  $\lceil (2n + 1)/3 \rceil$  distinct processes q: p received properly formed and justified
       $\langle (estimate, q, r_p, e_q, ts_q)_q, confirm_{s_q} \rangle_q$  from q];
     $estimate_{s_p} \leftarrow \{ \langle (estimate, q, r_p, e_q, ts_q)_q : p \text{ received properly formed and justified} \}$ 
       $\langle (estimate, q, r_p, e_q, ts_q)_q, confirm_{s_q} \rangle_q \text{ from } q \}$ ;
     $ts \leftarrow \text{largest } ts_q : \langle (estimate, q, r_p, e_q, ts_q)_q \in estimate_{s_p} \}$ ;
    if [ $ts = 0$  and (for  $\lceil (n - 1)/3 \rceil + 1$  distinct processes q and common value e:
       $\langle (estimate, q, r_p, e, ts)_q \in estimate_{s_p} \rangle$ ] then
       $es \leftarrow e$ ;
    else  $es \leftarrow e_q : \langle (estimate, q, r_p, e_q, ts)_q \in estimate_{s_p} \}$ ;
    send  $\langle (select, p, r_p, es, ts)_p, estimate_{s_p} \rangle_p$  to all;

  /* Phase 3: */
  wait until [(for  $\lceil (2n + 1)/3 \rceil$  distinct processes q and common value e: p received properly
    formed and justified  $\langle (confirm, q, r_p, e)_q, select_q \rangle_q$  from q) or  $c_p \in \mathcal{D}_p$ ];
  if [for  $\lceil (2n + 1)/3 \rceil$  distinct processes q and common value e: p received properly
    formed and justified  $\langle (confirm, q, r_p, e)_q, select_p \rangle_q$ ] then
     $ts_p \leftarrow r_p$ ;
     $e_p \leftarrow e$ ;
     $confirm_{s_p} \leftarrow \{ \langle (confirm, q, r_p, e)_q : p \text{ received properly formed and justified} \}$ 
       $\langle (confirm, q, r_p, e)_q, select_q \rangle_q \}$ ;
    send  $\langle (ready, p, r_p, e)_p, confirm_{s_p} \rangle_p$  to all;

/* Task 2: Confirmation */
repeat forever
  if [(p received properly formed and justified  $\langle (select, c', r', e', ts')_{c'}, estimate_{s_{c'}} \rangle_{c'}$ 
    from  $c' \equiv (r' \bmod n) + 1$ ) and
    (p has not previously sent  $\langle (confirm, p, r', -)_p, select_p \rangle_p$ )] then
     $select_p \leftarrow \{ \langle (select, c', r', e', ts')_{c'} \}$ ;
    send  $\langle (confirm, p, r', e')_p, select_p \rangle_p$  to all;

/* Task 3: Decision */
wait until [for  $\lceil (2n + 1)/3 \rceil$  distinct processes q and a common  $r', e'$ : p received
  properly formed and justified  $\langle (ready, q, r', e')_q, confirm_{s_q} \rangle_q$ ];
decide( $e'$ );

```

Figure 2: An algorithm to solve consensus using any fault detector \mathcal{D} in $\diamond\mathcal{S}(\text{Byz})$.

signature on the message is not valid, then the process does not accept the message. A process that receives a message must also determine that the message is *properly formed* and *properly justified*, as described below. If the message is not properly formed or is not properly justified, then the process does not accept the message.

Each message is structured as a *statement* and a *justification*. The statement represents the action of the process and is signed by the process. The justification is a set of statements with their signatures, extracted from messages received by the process, that together justify the statement. The entire message, containing both signed statement and justification, is signed by the process that originates it. Clever use of message digests and digital signatures allow a single signature per message, as described in the full version of the paper [6].

The *proper form* and *proper justification* of each type of message used by the consensus algorithm are now specified. The proper form of an *estimate* message sent by process p for round r_p is $\langle\langle estimate, p, r_p, e_p, ts_p \rangle_p, confirms_p \rangle_p$. The $confirms_p$ field is the justification of the *estimate* statement. This field is empty if $ts_p = 0$ and, otherwise, contains the signed $\lceil (2n + 1)/3 \rceil$ *confirm* statements that caused p to update its estimate to e_p and its timestamp to ts_p . The *confirm* statements contained in the $confirms_p$ field must all contain round ts_p and value e_p .

The proper form of a *select* message sent by coordinator c for round r_c is $\langle\langle select, c, r_c, es, ts \rangle_c, estimates_c \rangle_c$. The $estimates_c$ field is the justification of the *select* statement. It contains the signed $\lceil (2n + 1)/3 \rceil$ *estimate* statements that the coordinator c received. The identifier of the coordinator must satisfy $c = (r_c \bmod n) + 1$. The values es and ts in the *select* statement are determined by the corresponding values in the *estimate* statements.

The proper form of a *confirm* message sent by process p for round r_p is $\langle\langle confirm, p, r_p, e \rangle_p, select_p \rangle_p$. The $select_p$ field is the justification of the *confirm* statement and contains the *select* statement that p received from the coordinator. The *select* statement contained in the $select_p$ field must contain round r_p and value e .

The proper form of a *ready* message sent by process p for round r_p is $\langle\langle ready, p, r_p, e \rangle_p, confirms_p \rangle_p$. The $confirms_p$ field is the justification of the *ready* statement. It contains the signed $\lceil (2n + 1)/3 \rceil$ *confirm* statements that p received. All of the *confirm* statements contained in the $confirms_p$ field must contain round r_p and value e .

For the consensus algorithm described above, we have the following theorem:

Theorem 2. The consensus algorithm of Figure 2 solves the consensus problem using any fault detector in $\diamond\mathcal{S}(\text{Byz})$ in an asynchronous distributed system in which the maximum number k of Byzantine faults satisfies $k \leq \lfloor (n - 1)/3 \rfloor$.

The proof of Theorem 2 can be found in the full version of the paper [6]. By Theorems 1 and 2, we have the following corollary:

Corollary 1. Consensus is solvable using any fault detector in $\diamond\mathcal{W}(\text{Byz})$ in an asynchronous distributed system in which the maximum number k of Byzantine faults satisfies $k \leq \lfloor (n - 1)/3 \rfloor$.

The consensus algorithm is designed to mask most forms of Byzantine faults, but it depends on the fault detector to detect Byzantine coordinators that, by not sending

properly formed *select* messages or by sending different (mutant) *select* messages, attempt to block the algorithm in task 1 (Selection), phase 3. The consensus algorithm satisfies the safety properties of validity, agreement and irrevocability even if the fault detector fails to satisfy the completeness and accuracy properties. However, in such a case the consensus algorithm may fail to satisfy the liveness property of termination.

5 Latency Degree

In [13] Schiper introduced the notion of *latency degree* for measuring the cost of a distributed algorithm. The latency degree is defined by considering Lamport's logical clock [8] and the following rules:

- A *send* event and a *local* event at a process p do not modify p 's logical clock value
- If $ts(send(m))$ is the timestamp of the *send*(m) event and $ts(m)$ is the timestamp within a message m , then $ts(m) = ts(send(m)) + 1$
- If $ts(receive(m))$ is the timestamp of the *receive*(m) event at a process p , then $ts(receive(m))$ is the maximum of $ts(m)$ and the timestamp of the event at p immediately preceding the *receive*(m) event.

The *latency* of a run of a consensus algorithm is the largest timestamp of all decide events. The latency degree of a consensus algorithm is the minimal latency over all possible runs, which is typically obtained in a run in which no suspicions are generated. In [13] it is noted that the algorithm of Chandra and Toueg [2] that solves consensus using $\diamond\mathcal{S}$ has a latency degree of 4 and can be reduced to a latency degree of 3 by a trivial optimization.

With this measure, our consensus algorithm has a latency degree of 4. In contrast, the consensus algorithm of Malkhi and Reiter [10] has a latency degree of 9, if the reliable broadcast service is implemented using the algorithm in [11], and a latency degree of 6, if the reliable broadcast service is implemented using the algorithm of [9].

6 Implementing a Byzantine Fault Detector

In the literature, fault detectors are defined in terms of abstract properties rather than a specific implementation. However, the question of how to support such an abstraction in a real system must be addressed. Unreliable fault detectors for crash faults typically depend on timeouts; Byzantine faults are more varied and are more difficult to detect.

In a completely asynchronous system, it is impossible to implement a fault detector in $\diamond\mathcal{S}$ or $\diamond\mathcal{W}$. Such a fault detector could be used to solve consensus in an asynchronous system that is subject to faults, which has been shown to be impossible [4]. However, many practical systems can be expected to exhibit reasonable behavior most of the time, *e.g.*, there is some time after which the system stabilizes with bounds on relative process speeds and on message transmission times [3]. A fault detector in $\diamond\mathcal{S}$ or $\diamond\mathcal{W}$ can be implemented under such assumptions of partial synchrony. Similarly, in a completely asynchronous system it is impossible to implement a fault detector in $\diamond\mathcal{S}(\text{Byz})$ or $\diamond\mathcal{W}(\text{Byz})$. However, if we make similar assumptions with regard to relative speeds of correct processes and message transmission times, and

if we assume that signatures are unforgeable, then a Byzantine fault detector can be implemented.

We describe here an implementation of a Byzantine fault detector that can be used in conjunction with the consensus algorithm presented in Section 4. The local fault detector module at process p provides an output consisting of a list of processors that p currently suspects of having exhibited a detectable Byzantine fault, which is distinguished from non-detectable Byzantine faults as defined below.

Non-detectable Byzantine faults are (1) faults that are unobservable by processes based on the messages they receive, and (2) faults that are undiagnosable, *i.e.*, cannot be attributed to a particular process. For example, if a Byzantine process q sends to all processes an initial *estimate* message containing a value that is different from its internal input value, that behavior is unobservable. Also, if a Byzantine process q sends a message that purportedly was sent by a process p but that does not contain a valid signature then, in the absence of further information, it is impossible to determine whether q sent the message, and so that fault is undiagnosable.

Detectable Byzantine faults are classified as omission faults and commission faults. An omission fault occurs when a process omits to send a required message to one or more correct processes.

There are two types of commission faults: (1) a process q sends a message that is not properly formed or properly justified, according to the definitions in Section 4, and (2) a process sends two or more *mutant* messages, each of which is properly formed and individually justified, but which have the same type, the same source, and the same round but different contents. If a process receives a message that is not properly formed or properly justified, or receives two mutant messages, then the process can detect that the sender of those messages is Byzantine.

For our algorithm, the most critical instance of a *commission fault of type 2* occurs if the coordinator sends a *select* message m with value e to some correct process p and sends a mutant *select* message m' with value e' to some correct process q .⁴ If this occurs, the coordinator is Byzantine, and the liveness of the consensus algorithm depends on the detection that the coordinator is faulty by all of the correct processes.

The detection occurs as follows. According to the algorithm, every correct process that receives a *select* message sends, to every process, a *confirm* message that contains the *select* statement as justification. Thus, p will send, to all processes, a *confirm* message containing as justification the *select* statement with value e that it has received. Similarly, q will send, to all processes, a *confirm* message containing as justification the *select* statement with value e' that it has received. Thus, all correct processes will receive both the *select* statement with value e and the *select* statement with value e' . Any correct process that receives two mutant *select* statements, directly or indirectly, will declare the coordinator to be Byzantine.

The local fault detector module at a process p monitors messages that are sent and received by the consensus algorithm, and produces as output a list $output_p$ of suspects. The fault detector makes use of timeouts. When p sets a timeout for round r_p , an interval begins during which p expects to receive a certain set of messages for

⁴For both *select* messages to be properly formed and justified, both must contain timestamp 0.

round r_p . This interval is measured in ticks on p 's local clock. Once a timeout is set, it can either *expire* or be *canceled*. The timeout expires if p has not received the expected messages by the end of the interval. The timeout is canceled if p receives the expected messages before the end of the interval.

It is impossible to know with certainty whether an omission fault has occurred or a process is simply slow or a message has been delayed. However, if an improperly formed or justified message, or two mutant messages, are received by a correct process, then that process knows with certainty that a commission fault has occurred (assuming that signatures are unforgeable). For example, if a process p receives properly signed mutant messages m and m' from a process q , then p has proof that q has committed a commission fault of type 2. In such a case, q is known to be Byzantine and should be permanently suspected by p . To implement this, the fault detector at p maintains a list $byzset_p$ of processes that p knows to be Byzantine.

The fault detector algorithm consists of five concurrent tasks, shown in Figure 3. In the first task, when p sends an *estimate* message to the coordinator of round r_p , p sets a timeout for round r_p .

In the second task, when a timeout expires for round r_p , p adds the coordinator c_p of round r_p to its list $output_p$, containing the identifiers of the processes it suspects, and adds r_p to its list $timeouts_p$, containing the round numbers for which timeouts have expired.

In the third task, when p receives *confirm* messages with the same value for round r_p from $\lceil(2n + 1)/3\rceil$ distinct processes, p cancels the timeout for round r_p if it has not yet expired. However, if the timeout has already expired (indicating that p has prematurely timed out for round r_p), then p removes r_p from $timeouts_p$ and checks to see if the coordinator c_p of round r_p should be removed from its suspects list $output_p$. If c_p is not in $byzset_p$ and there is no round number in $timeouts_p$ for which c_p was the coordinator, then p removes c_p from $output_p$.

In the fourth task, when p receives a signed message m from q that is not properly formed or properly justified according to the definitions in Section 4, p adds q to its suspects list $output_p$. Additionally, p adds q to its list $byzset_p$ to indicate that q should never subsequently be removed from $output_p$.

In the fifth task, when p directly or indirectly receives a message m signed by q that is a mutant of a message m' that p has already received, p adds q to $output_p$ and to $byzset_p$.

There is a subtle point that arises with regard to task 3. In task 3, p waits to receive *confirm* messages from $\lceil(2n + 1)/3\rceil$ distinct processes, but if the timeout expires, it is the coordinator c_p that p will suspect, even if p received a *select* message from c_p , and even if p received a *confirm* message from c_p . The reason is the following. If the coordinator c_p is Byzantine, it may send a *select* message to a proper subset of correct processes rather than to all of them. If c_p sends a *select* message to fewer than $\lceil(2n + 1)/3\rceil$ correct processes, then some correct process p may not be able to collect *confirm* messages from $\lceil(2n + 1)/3\rceil$ processes. Therefore, p will wait in phase 3 of task 1 of the consensus algorithm until p comes to suspect the coordinator c_p . If p 's timeout expires before the *confirm* messages are collected, then in task 3 of the fault detector p will come to suspect c_p , and will not wait forever in phase 3 of task 1 of

/* Each process p executes the following */	
/* Initialization */	
$output_p \leftarrow \emptyset;$	/* The set of processes suspected by p */
$byzset_p \leftarrow \emptyset;$	/* The set of processes known by p to be Byzantine */
$timeouts_p \leftarrow \emptyset;$	/* The set of round numbers for which timeouts have expired */
initiate concurrent tasks 1, 2, 3, 4 and 5;	
/* Task 1: */	
when [p sends $\langle (estimate, p, r_p, e_p, ts_p)_p, confirms_p \rangle$ to c_p]	
set timeout for round r_p ;	
/* Task 2: */	
when [p 's timeout for round r_p expires]	
$timeouts_p \leftarrow (timeouts_p \cup \{r_p\});$	
$output_p \leftarrow (output_p \cup \{c_p\}) : c_p \equiv (r_p \bmod n) + 1;$	
/* Task 3: */	
when [for $\lceil (2n + 1)/3 \rceil$ distinct processes q and common value e : p received properly formed and justified $\langle (confirm, q, r_p, e)_q, select_q \rangle_q$ from q]	
if [$r_p \notin timeouts_p$] then	
cancel timeout for round r_p ;	
else /* Premature timeout has occurred for round r_p */	
$timeouts_p \leftarrow (timeouts_p - \{r_p\});$	
if [for $c_p \equiv (r_p \bmod n) + 1$ and for all $r \in timeouts_p$:	
$c_p \notin byzset_p$ and $c_p \neq (r \bmod n) + 1$] then	
$output_p \leftarrow (output_p - \{c_p\});$	
/* Task 4: */ /* Commission fault of type 1 */	
when [p receives a message from q that is not properly formed or is not properly justified]	
$output_p \leftarrow (output_p \cup \{q\});$	
$byzset_p \leftarrow (byzset_p \cup \{q\});$	
/* Task 5: */ /* Commission fault of type 2 */	
when [p receives a message m from q that is a mutant of a previous message m' from q]	
$output_p \leftarrow (output_p \cup \{q\});$	
$byzset_p \leftarrow (byzset_p \cup \{q\});$	

Figure 3: An implementation of a Byzantine fault detector in a model of partial synchrony and unforgeable signatures that can be used in conjunction with the consensus algorithm.

the consensus algorithm. On the other hand, if the coordinator is correct, then each correct process will eventually receive the *select* message and will eventually send a *confirm* message; thus, each correct process will eventually receive *confirm* messages from $\lceil (2n + 1)/3 \rceil$ distinct processes.

If a *confirm* message sent by a correct process q is delayed in reaching another correct process p , either because q is slow or because transmission of the message is slow, then p may incorrectly come to suspect the coordinator. However, this mistake by the fault detector is allowed by the properties of the fault detector. Eventually, when p receives the message, it will remove c_p from its list of suspects $output_p$.

The fault detector does not detect non-detectable Byzantine faults, but the consensus algorithm satisfies the properties of termination, validity, agreement and irrevocability even though the fault detector does not detect such faults, provided that there are no more than $\lfloor (n - 1)/3 \rfloor$ Byzantine faults.

The fault detector algorithm of Figure 3 has been specifically designed to work with the consensus algorithm of Figure 2. A more general Byzantine fault detector, such as that described in [7], could instead be used. The fault detector of [7] makes use of a message diffusion algorithm to ensure that, if a Byzantine process sends two mutant messages to two different correct processes, then all correct processes will receive both mutant messages and thus will declare the sender to be Byzantine. The message diffusion algorithm also masks Byzantine faults in which a Byzantine process sends a message to a subset of the correct processes, rather than to all of the processes. The fault detector of [7] makes use of timeouts that are set for every required message.

7 The Weakest Byzantine Fault Detectors

In [1] Chandra, Hadzilacos and Toueg have demonstrated that the fault detectors in the class $\diamond\mathcal{W}$ are the weakest fault detectors that allow consensus to be solved in an asynchronous distributed system in which the maximum number f of crash faults satisfies $f \leq \lfloor (n-1)/2 \rfloor$. They prove this by showing that any fault detector that can be used to solve consensus in such an environment can be transformed into a fault detector in $\diamond\mathcal{W}$. Their proof is elegant and ingenious but complex and lengthy.

As noted in Section 3, a fault detector in $\diamond\mathcal{W}$ is too weak to solve consensus in an asynchronous distributed system if Byzantine faults can occur. However, by Corollary 1, any fault detector in $\diamond\mathcal{W}(\text{Byz})$ allows consensus to be solved, provided that the maximum number k of Byzantine faults satisfies $k \leq \lfloor (n-1)/3 \rfloor$. Moreover, we have the following theorem:

Theorem 3. The class $\diamond\mathcal{W}(\text{Byz})$ of unreliable Byzantine fault detectors is the class of weakest fault detectors that allow consensus to be solved in an asynchronous distributed system that is subject to k Byzantine faults, where $k \leq \lfloor (n-1)/3 \rfloor$.

The proof of Theorem 3 is based on the proof in [1], and is briefly sketched below. More details can be found in the full version of the paper [6].

The objective of the proof is to show that any fault detector \mathcal{D} that can be used to achieve consensus in an asynchronous distributed system with no more than $\lfloor (n-1)/3 \rfloor$ Byzantine faults can be transformed into a fault detector in $\diamond\mathcal{W}(\text{Byz})$. The transformation uses the algorithm $\text{Consensus}_{\mathcal{D}}$ that achieves consensus using the fault detector \mathcal{D} .

The proof utilizes a class $\Omega(\text{Byz})$ of intermediate fault detectors that report only a single correct process. The definition of the $\Omega(\text{Byz})$ class of fault detectors is identical to that of the Ω class of Chandra, Hadzilacos and Toueg, and is stated in terms of the following property: There is a time after which all correct processes permanently trust the same correct process.

A fault detector in $\Omega(\text{Byz})$ is readily transformed into a fault detector in $\diamond\mathcal{S}(\text{Byz})$. The algorithm that transforms a fault detector in $\Omega(\text{Byz})$ into a fault detector in $\diamond\mathcal{S}(\text{Byz})$ simply sets output_p equal to all of the processes except the process that p currently trusts according to the fault detector in $\Omega(\text{Byz})$. The $\diamond\mathcal{S}(\text{Byz})$ class is equivalent to the $\diamond\mathcal{W}(\text{Byz})$ class; thus, a fault detector in $\Omega(\text{Byz})$ can be transformed into a fault detector in $\diamond\mathcal{W}(\text{Byz})$.

The bulk of the proof consists of showing that the given fault detector \mathcal{D} can be transformed into a fault detector in $\Omega(\text{Byz})$ using $\text{Consensus}_{\mathcal{D}}$. By transitivity, it then follows that \mathcal{D} can be transformed into a fault detector in $\diamond\mathcal{W}(\text{Byz})$. The transformation implies that $\text{Consensus}_{\mathcal{D}}$ and \mathcal{D} taken together provide at least as much information about faults as a fault detector in $\Omega(\text{Byz})$. Note that some of the information necessary for the detection of faults may be provided by the consensus algorithm itself.

In fact, the fault detector of Figure 3 does not actually provide the properties of $\diamond\mathcal{S}(\text{Byz})$ (even in a model of partial synchrony) in that it does not detect all possible instances of detectable Byzantine behavior. However, the consensus algorithm of Figure 2 and the fault detector of Figure 3 work together to provide at least as much information about faults as a fault detector in $\Omega(\text{Byz})$. In general, any consensus algorithm may mask or detect some faults. The specific implementation of the fault detector that works together with the consensus algorithm must provide the rest of the information about faults.

In particular, the $\diamond\mathcal{S}(\text{bz})$ class of Malkhi and Reiter [10] may appear to be weaker than the $\diamond\mathcal{S}(\text{Byz})$ class presented here, as a fault detector in $\diamond\mathcal{S}(\text{bz})$ only detects quiet behavior. However, Malkhi and Reiter ascribe detection of some types of Byzantine faults other than quiet behavior to their consensus algorithm, which adds processes that send non-well-formed messages (defined as malformed, out-of-order, or unjustifiable messages) to the suspects list. Without a means of detecting non-well-formed messages, a correct process could wait forever for a particular message from a process that is faulty but that is not quiet. Thus, their mechanism that detects non-well-formed messages provides information about faults, and must be considered in combination with the fault detector when considering the strength of the fault detector.

8 Conclusion

We have extended the use of fault detectors for solving consensus to asynchronous distributed systems that are subject to Byzantine faults by defining two classes of unreliable Byzantine fault detectors: the $\diamond\mathcal{S}(\text{Byz})$ class and the $\diamond\mathcal{W}(\text{Byz})$ class. We have given a transformation algorithm that transforms any fault detector in $\diamond\mathcal{W}(\text{Byz})$ into a fault detector in $\diamond\mathcal{S}(\text{Byz})$. Moreover, we have given an algorithm that solves consensus in an asynchronous distributed system that is subject to at most $\lfloor (n-1)/3 \rfloor$ Byzantine faults, using a fault detector in $\diamond\mathcal{S}(\text{Byz})$. Thus, a fault detector in $\diamond\mathcal{W}(\text{Byz})$ can also be used to solve consensus in an asynchronous system that is subject to Byzantine faults. Furthermore, any fault detector that can be used to solve consensus in such an environment must be at least as strong as a fault detector in $\diamond\mathcal{W}(\text{Byz})$.

Acknowledgments

We wish to thank the anonymous referees for their constructive comments, which have greatly improved this paper.

References

- [1] T. D. Chandra, V. Hadzilacos and S. Toueg, “The weakest failure detector for solving consensus,” *Journal of the ACM*, vol. 43, no. 4 (July 1996), pp. 685-722.
- [2] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM*, vol. 43, no. 2 (March 1996), pp. 225-267.
- [3] C. Dwork, N. Lynch and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM*, vol. 35, no. 2 (April 1988), pp. 288-323.
- [4] M. J. Fischer, N. A. Lynch and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM*, vol. 32, no. 2 (April 1985), pp. 374-382.
- [5] R. Guerraoui and A. Schiper, “‘T-accurate’ failure detectors,” *Proceedings of the 10th International Workshop on Distributed Algorithms*, Bologna, Italy (October 1996), Lecture Notes in Computer Science 1151, pp. 269-286.
- [6] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, “Unreliable Byzantine fault detectors for solving consensus,” Department of Electrical and Computer Engineering, University of California, Santa Barbara, Technical Report 97-14.
- [7] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith, “Implementing an unreliable Byzantine fault detector,” Department of Electrical and Computer Engineering, University of California, Santa Barbara, Technical Report 97-15.
- [8] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7 (July 1978), pp. 558-565.
- [9] D. Malkhi and M. Reiter, “A high-throughput secure reliable multicast protocol,” *Proceedings of the 9th Computer Security Foundations Workshop*, Kenmore, Ireland (June 1996), pp. 9-17.
- [10] D. Malkhi and M. Reiter, “Unreliable intrusion detection in distributed computations,” *Proceedings of the 10th Computer Security Foundations Workshop*, Rockport, MA (June 1997), pp. 116-124.
- [11] M. K. Reiter, “Secure agreement protocols: Reliable and atomic group multicast in Rampart,” *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (November 1994), pp. 68-80.
- [12] R. L. Rivest, A. Shamir and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2 (February 1978), pp. 120-126.
- [13] A. Schiper, “Early consensus in an asynchronous system with a weak failure detector,” *Distributed Computing*, vol. 10 (1997), pp. 149-157.