Byzantine Fault Detectors for Solving Consensus*

KIM POTTER KIHLSTROM[1], LOUISE E. MOSER[2] AND P. M. MELLIAR-SMITH[2]

[1]*Department of Mathematics and Computer Science, Westmont College, 955 La Paz Road, Santa Barbara, CA 93108, USA*
[2]*Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA*
Email: kimkihls@westmont.edu

Unreliable fault detectors can be defined in terms of completeness and accuracy properties and 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. First, we define and categorize Byzantine faults. We then define two new completeness properties, eventual strong completeness and eventual weak completeness. We use these completeness properties and previously defined accuracy properties to define four new classes of unreliable Byzantine fault detectors. Next, we present an algorithm that uses a Byzantine fault detector to solve the consensus problem in an asynchronous distributed system of n processes in which the number k of Byzantine faults satisfies $k \le \lfloor (n-1)/3 \rfloor$. We also give algorithms that implement a Byzantine fault detector in a model of partial synchrony. Finally, we prove the correctness of the consensus algorithm and analyze its complexity.

Received 6 August 2001; revised 12 August 2002

1. INTRODUCTION

In a distributed system, it is often necessary for the processors to reach agreement or consensus. For example, in a distributed database system, the processors need to reach agreement on whether a transaction should be committed or aborted. In a system that involves replicated data, the processors need to reach agreement about the delivery order for messages that contain operations on the data. In a fault-tolerant distributed system, processors need to reach agreement on which of the processors are currently operational. The consensus problem is a fundamental problem in distributed computing that is related to all of these agreement problems. Fischer *et al.* [2] 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 [3] have shown, however, that consensus is possible in an asynchronous distributed system that is subject to crash faults, if the asynchronous model is augmented with an unreliable fault detector.

In this paper we extend the work of Chandra and Toueg [3] by considering unreliable fault detectors for solving consensus in an asynchronous distributed system that is subject to Byzantine faults. A Byzantine processor is a processor that behaves in an arbitrary manner, or even according to some malicious design. It might send, to different destinations, different messages that purport to be the same message. We call such messages *mutant* messages. Furthermore, Byzantine processors might collaborate in an attempt to bring the system down. We use the notion of a Byzantine fault to model a processor in which the software has been corrupted due to a malicious intrusion or a Trojan horse, or in which the software has become confused due to a programming error, or to a processor that behaves in an arbitrary way due to random bit flips, as occur in space applications due to bombardment by cosmic rays.

We capture the essence of Byzantine faults by defining them in terms of deviation from the algorithm A that the processes execute and give a categorization of Byzantine faults. We define two new completeness properties, eventual strong Byzantine completeness for algorithm A and eventual weak Byzantine $(k+1)$-completeness for algorithm A. We use these completeness properties and previously defined accuracy properties [3] to define four new classes of unreliable Byzantine fault detectors. Then we present an algorithm that uses a Byzantine fault detector to solve the consensus problem in an asynchronous distributed system with n processes of which at most $\lfloor (n-1)/3 \rfloor$ are Byzantine. We also give algorithms that implement a Byzantine fault detector in a model of partial synchrony. Finally, we prove the correctness of our consensus algorithm and analyze its complexity.

2. RELATED WORK

Chandra and Toueg [3] have identified two critical properties of unreliable fault detectors: completeness and accuracy.

*A preliminary version of this paper appeared in the *Proceedings of the International Conference on Principles of Distributed Systems* [1].

These properties include the following variations.

- *Eventual strong completeness*. There is a time after which every process that crashes is permanently suspected by *every* correct process.
- *Eventual weak completeness*. There is a time after which every process that crashes is permanently suspected by *some* correct process.
- *Eventual strong accuracy*. There is a time after which *every* correct process is never suspected by any correct process.
- *Eventual weak accuracy*. There is a time after which *some* correct process is never suspected by any correct process.

Chandra and Toueg have defined several classes of unreliable fault detectors in terms of these completeness and accuracy properties. For example, their $\diamond\mathcal{S}$ class satisfies eventual strong completeness and eventual weak accuracy, whereas their $\diamond\mathcal{W}$ class satisfies eventual weak completeness and eventual weak accuracy. Chandra and Toueg have given an algorithm that uses any fault detector in $\diamond\mathcal{S}$ to solve consensus in an asynchronous distributed system in which the number f of crash faults satisfies $f < \lceil n/2 \rceil$. Chandra *et al.* [4] have proved that the class $\diamond\mathcal{W}$ contains the weakest fault detectors that can be used to solve consensus in such an asynchronous distributed system.

Dwork *et al.* [5] have shown that, in a model of partially synchronous communication, a solution to the consensus problem is possible in the presence of Byzantine faults if and only if the number k of Byzantine faults satisfies $k \leq \lfloor (n-1)/3 \rfloor$, where n is the number of processes in the system. This result holds whether or not digital signatures are employed. The fault detectors presented here (and by Chandra and Toueg [3]) can be implemented if communication is partially synchronous. Consequently, in our model, consensus can be solved if and only if the number k of Byzantine faults satisfies $k \leq \lfloor (n-1)/3 \rfloor$.

Malkhi and Reiter [6] have defined a class $\diamond\mathcal{S}(\text{bz})$ of eventually strong fault detectors and have shown that a fault detector in $\diamond\mathcal{S}(\text{bz})$ can be used to solve the binary consensus problem in an asynchronous distributed system that is subject to at most $\lfloor (n-1)/3 \rfloor$ Byzantine faults. Their fault detector is defined in terms of a source ordered, causally ordered reliable broadcast service. The fault detector of Malkhi and Reiter detects only *quiet* behavior, where a process is quiet if some correct process receives only a finite number of reliable broadcasts from that process in an infinite run. Malkhi and Reiter ascribe detection of some Byzantine faults to the consensus algorithm, which adds processes that send *non-well-formed* messages (defined as messages that are malformed, out-of-order, or unjustified) to the suspects list.

Doudou and Schiper [7] have defined fault detectors for a Byzantine environment based on the notion of *mute* processes. Mute processes are similar to the quiet processes presented by Malkhi and Reiter, but are not defined in terms of a reliable broadcast algorithm. A mute process is defined as a process that crashes, that ceases sending any messages, or that sends only unsigned messages. Doudou and Schiper take the approach of Malkhi and Reiter regarding the types of Byzantine behavior that are detected by the fault detector. Their fault detector detects only behavior that disrupts progress; the handling of all other Byzantine behavior is deferred to the consensus protocol. Doudou and Schiper have described a vector validity property in which processes decide on a vector of size n (where n is the number of processes in the system) rather than on a single value. Doudou *et al.* [8] have defined failure detectors for an environment subject only to *mute* failures. The muteness failure model is defined to be weaker than the Byzantine failure model; a mute process stops sending algorithm messages but might continue sending other messages.

In the models of Malkhi and Reiter and of Doudou and Schiper, some information about Byzantine faults that could be made available is not used for fault detection, but rather is masked by the reliable broadcast service or by the consensus algorithm. We take a different approach and attempt to capture as much information about Byzantine faults as possible. Capturing as much information about Byzantine faults as possible, rather than masking such faults, allows faulty processes to be detected and excluded from the system. Once a faulty process has been removed, it no longer counts against the resilience requirement. Capturing as much information about Byzantine faults as possible is particularly necessary if Byzantine fault detectors are to be used to solve other consensus-related problems, such as the group membership problem.

Alvisi *et al.* [9] have described statistical techniques for detecting Byzantine server failures. Their work differs from ours in that it focuses on detecting Byzantine faults in the context of replicated data using Byzantine quorum systems, without regard to failure detector specifications or the consensus problem.

3. 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. As described below, the asynchronous system is augmented with a fault detector.

The communication network consists of communication channels on which pairs of processes communicate via messages using the *send* and *receive* primitives. 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. However, messages may be delayed and may be delivered in a different order than the one in which they were sent.

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, crash faults are included among the Byzantine faults that 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.

Messages can be *broadcast* to all n processes using multiple send operations. However, a Byzantine process can broadcast two messages with different contents to different subsets of the processes or can broadcast a message to only a proper subset of the processes.

We assume that an authentication mechanism is available such that a process can verify the original sender of a message, even if another process has relayed that message. Thus, our model is the authenticated Byzantine model. We use a public key cryptosystem such as RSA [10] 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 $\langle - \rangle_p$.[3] 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 means of key distribution is available such that all correct processes have the same public key for each of the processes. We also employ a message digest function such as MD5 [11] in which an arbitrary length input message m is mapped to a fixed length output $d(m)$. We assume that a message digest uniquely identifies a message.

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 variations of the consensus problem. The consensus problem that we consider is the binary consensus problem in which the input values and the decision value are binary. The specification of this problem is as follows.

- *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.

We describe Byzantine fault detectors that augment the asynchronous system. 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. In a completely asynchronous system, it is impossible to implement a (crash) fault detector in $\diamond\mathcal{S}$ or $\diamond\mathcal{W}$. Such a fault detector could be used to solve consensus [3] in an asynchronous system that is subject to faults, which has been shown to be impossible [2].

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 processing and message transmission times. A (crash) fault detector in $\diamond\mathcal{S}$ or $\diamond\mathcal{W}$ can be implemented under these assumptions of partial synchrony [3]. Similarly, in a completely asynchronous system it is impossible to implement a Byzantine fault detector such as we describe. However, if we make similar assumptions with regard to message transmission times and the processing times of correct processes, then a Byzantine fault detector can be implemented.

For the purpose of implementing the fault detectors that we describe, we assume the $\mathcal{M}_3$ model of partial synchrony described by Chandra and Toueg [3]. In this model, there are bounds on processing and message transmission times, but these bounds are not known and they only hold after some unknown time, called the *global stabilization time* (GST). Although it is convenient to assume that the bounds hold *forever* after GST, in practice it is sufficient that the bounds hold only long enough to make progress.

4. CATEGORIZING BYZANTINE FAULTS

In a system in which processes can fail only by crashing, fault detection can be accomplished as follows [3]. Each process periodically sends an 'I am alive' message to all of the other processes. If a process p has not received such a message from another process q during a specified time interval, p begins to suspect that q is faulty. Chandra and Toueg [3] have given an algorithm that uses this method to implement, in a model of partial synchrony, a fault detector that satisfies eventual strong accuracy and eventual strong completeness. That approach does not work for systems in which Byzantine faults can occur. A Byzantine process could, for example, faithfully send periodic 'I am alive' messages, while at the same time refuse to send any messages required by the algorithm that uses the fault detector.

We therefore define Byzantine faults in terms of deviation from the algorithm A that the processes execute. However, we cannot detect faults that are evidenced only in the internal state of a Byzantine process, because the external behavior of a Byzantine process can be inconsistent with its internal state. For example, a Byzantine process executing a consensus algorithm can inaccurately report its input value, or can decide on a value other than the consensus value.

We distinguish between *detectable* and *non-detectable* Byzantine faults. Our classification of Byzantine faults is illustrated in Figure 1. 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 p sends to all processes a message claiming that its initial value is something other than its internal input value, that behavior is unobservable. Also, if a Byzantine process p sends a message that purportedly was sent by process q but that does not contain a valid signature then, in the absence of further information, that fault is undiagnosable.

[3]Throughout this paper, we use – to refer to an arbitrary value of an appropriate type.

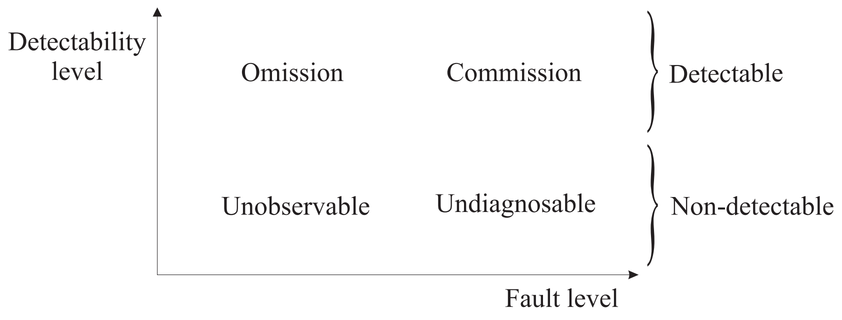

FIGURE 1. Classification of Byzantine faults.

Detectable Byzantine faults refer to deviations that are evidenced in the external behavior of a process. Detectable deviation from an algorithm is defined in terms of the messages sent (or not sent) by processes during a particular execution of the algorithm.

We define detectable deviation from an algorithm A in terms of *omission* faults and *commission* faults. An omission fault occurs when a process fails to send a message that it should have sent in an execution E of algorithm A. Although we desire to capture as much information about faults as possible, in a Byzantine environment we are unable to ensure that correct processes will detect the omission of a single point-to-point message. We discuss this further in Section 8. Thus, for a process p and an execution E of algorithm A, an omission fault occurs when:

- no correct process ever receives m, where m is a message that p is required to send during execution E of algorithm A.

A commission fault occurs when a process sends a message that it should not have sent during execution E of algorithm A. For a process p and an execution E of algorithm A, a commission fault occurs when:

- process p sends to some correct process a message m and sends to some correct process a mutant message m', where *mutant* messages are defined as two or more messages that have the same message header, but different contents and that each individually conform to algorithm A for execution E;
- process p sends to some correct process a message that could not have been sent by a correct process executing algorithm A, based on the messages p previously received during execution E.

These two types of commission faults can be viewed as violations of the restrictions specified by Bracha [12], Coan [13] and Neiger and Toueg [14]; namely: (1) a process must send the same message to all processes; and (2) the message must conform to the algorithm being executed.

5. DESCRIBING THE ALGORITHM A

We make certain assumptions with regard to the algorithm A that the processes run. First, we assume that algorithm A is live and deterministic in that it requires each correct

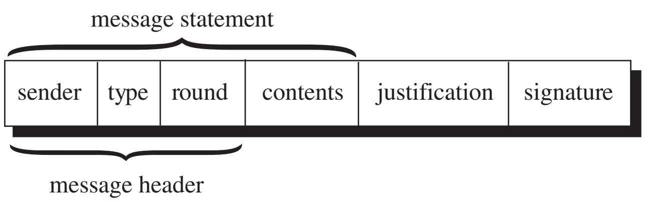

FIGURE 2. Sample message format.

process to send certain messages continuously; we refer to these as *required* messages. Furthermore, we assume that every message m sent by a correct process executing algorithm A is sent to all processes. We also require that, when a correct process receives a message for the first time, it must immediately relay the message to all processes.

We assume that every message m sent by a correct process executing algorithm A includes a message *header* that uniquely identifies the message. The message header consists of several fields, including the identity of the originator of the message, denoted *m.sender* and the message type, denoted *m.type*. Depending on the algorithm A, the message header may also include other fields such as a round number, sequence number, or timestamp. The header is defined as the part of the message that the receiver can anticipate.

In addition to the message header, each message also includes some information that is used by algorithm A, which we refer to as the message *contents*. Depending on the algorithm, this might include a value, vote, or operation. A sample message format is shown in Figure 2.

The header and contents of a message together are referred to as the message *statement*, as shown in Figure 2. The statement represents the action of the process and is signed by the process that sends it. In addition to the message header and contents, each message includes a *justification* field. The justification is a set of statements with their signatures, extracted from messages received by the process, that together justify the statement. Justification of messages has been used previously in the context of Byzantine faults [14, 15].

We assume that a correct process p executing algorithm A knows the type of message that it currently expects from another process executing algorithm A and can set an appropriate timeout for a message with the expected message header. For example, if algorithm A is a consensus algorithm such as that given in Figure 3, then a process p that is executing the algorithm can expect the coordinator q of round r to send a message that contains the appropriate message type, the identifier of q, the round number r and the value that q has selected for consensus. When p enables round r, it sets a timeout for a message with the given message header.

6. DEFINING UNRELIABLE BYZANTINE FAULT DETECTORS

We define Byzantine fault detectors in terms of detectable Byzantine faults, i.e. detectable deviation from the algorithm that uses the fault detector, as described in Section 4. We define a new completeness property for a Byzantine fault detector used by algorithm A as follows.

- *Eventual strong Byzantine completeness for algorithm A*. There is a time after which every process that has detectably deviated from algorithm A is permanently suspected by every correct process.

Some faults may be masked by algorithm A, but our basic approach is to require that the fault detector detects as many faults as possible. Thus, we detect all omission and commission faults as defined in Section 4.

In addition to the completeness property defined above, we also make use of the eventual accuracy properties defined by Chandra and Toueg [3] and restated in Section 2. We then define two classes of Byzantine fault detectors as follows. A Byzantine fault detector is in the class $\diamond\mathcal{P}(\text{Byz}, A)$ if it satisfies eventual strong Byzantine completeness for algorithm A and eventual strong accuracy. A Byzantine fault detector is in the class $\diamond\mathcal{S}(\text{Byz}, A)$ if it satisfies eventual strong Byzantine completeness for algorithm A and eventual weak accuracy.

Chandra and Toueg [3] have defined eventual strong and weak completeness properties for the crash fault model and have given a transformation algorithm that transforms a fault detector $\mathcal{D}$ that satisfies eventual weak completeness into a fault detector $\mathcal{D}'$ that satisfies eventual strong completeness. The algorithm uses $\mathcal{D}$ to maintain a variable $output_p$ that emulates the output of $\mathcal{D}'$. In the algorithm, each process periodically sends the suspects list obtained from its local fault detector module $\mathcal{D}_p$ to every process. When a process p receives a suspects list from another process q, it adds the suspects in the list to $output_p$ and removes process q from $output_p$.

We could attempt to define classes of Byzantine fault detectors in terms of the following eventual weak completeness property.

- There is a time after which every process that has detectably deviated from algorithm A is permanently suspected by some correct process.

However, in a system that is subject to Byzantine faults, although the transformation algorithm of Chandra and Toueg [3] transforms the above property into eventual strong Byzantine completeness for algorithm A, it cannot ensure that eventual weak or strong accuracy is preserved in the presence of even a single Byzantine fault. The problem is that a Byzantine process can report repeatedly that it suspects every process, thus causing the fault detector $\mathcal{D}'$ to report repeatedly that every process is faulty. Thus, even though the fault detector $\mathcal{D}$ satisfies eventual weak or strong accuracy, the fault detector $\mathcal{D}'$ does not.

TABLE 1. Classes of Byzantine fault detectors.

Completeness for algorithm A	Accuracy: Eventual strong	Accuracy: Eventual weak
Eventual strong Byzantine	*Eventually perfect* $\diamond\mathcal{P}(\text{Byz}, A)$	*Eventually strong* $\diamond\mathcal{S}(\text{Byz}, A)$
Eventual weak Byzantine $(k+1)$	$\diamond\mathcal{Q}(\text{Byz}, A)$	*Eventually weak* $\diamond\mathcal{W}(\text{Byz}, A)$

To address this difficulty, we could consider an alternative transformation algorithm in which a process p adds another process q to $output_p$ only if p has received reports from $k+1$ processes (possibly including itself) that regard q as faulty according to $\mathcal{D}$. While such an algorithm preserves eventual weak or strong accuracy, it does not transform the eventual weak completeness property given above into eventual strong Byzantine completeness for algorithm A. 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 required $k+1$ reports.

We therefore define a new weak completeness property for a fault detector used by algorithm A as follows.

- *Eventual weak Byzantine $(k+1)$-completeness for algorithm A*. There is a time after which every process that has detectably deviated from algorithm A is permanently suspected by at least $k+1$ correct processes.

We then define two more classes of Byzantine fault detectors as follows. A fault detector is in the class $\diamond\mathcal{Q}(\text{Byz}, A)$ if it satisfies eventual weak Byzantine $(k+1)$-completeness for algorithm A and eventual strong accuracy. A fault detector is in the class $\diamond\mathcal{W}(\text{Byz}, A)$ if it satisfies eventual weak Byzantine $(k+1)$-completeness for algorithm A and eventual weak accuracy. The four classes of Byzantine fault detectors are shown in Table 1.

7. SOLVING CONSENSUS

We now present an algorithm A_1 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 $\mathcal{D}_1$ in $\diamond\mathcal{W}(\text{Byz}, A_1)$. As part of the algorithm, $\mathcal{D}_1$ is transformed into a fault detector $\mathcal{D}_2$ in $\diamond\mathcal{S}(\text{Byz}, A_1)$. Pseudocode for the consensus algorithm A_1 is given in Figure 3.

```
/* Each process p executes the following */

/* Initialization */
$e_p \leftarrow v_p$;                          /* $e_p$ 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 $e_p$ */
$\text{confirms}_p \leftarrow \emptyset$;      /* $\text{confirms}_p$ is the set of CONFIRM messages that caused p to update $e_p$ */
$\text{suspected}_p \leftarrow \emptyset$;     /* $\text{suspected}_p$ is the set of processes suspected by p */
$\text{output}(\mathcal{D}_2)_p \leftarrow \emptyset$;   /* $\text{output}(\mathcal{D}_2)_p$ emulates $\mathcal{D}_{2p}$ */
for each $r$ in $S$
  $\text{suspecting}_p[r] \leftarrow \emptyset$;   /* the set of processes p 'thinks' are currently suspecting r */

cobegin                                        /* Five concurrent tasks */
  /* Task 1: */
  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 $\langle\langle \text{ESTIMATE}, p, r_p, e_p, ts_p\rangle_p, \text{confirms}_p\rangle_p$ to all;

    /* Phase 2: */
      if $[p = c_p]$ then
          wait until [for $n - k$ distinct processes $q$: $p$ received properly formed and justified
                       $\langle\langle \text{ESTIMATE}, q, r_p, e_q, ts_q\rangle_q, \text{confirms}_q\rangle_q$ from $q$];
          $\text{estimates}_p \leftarrow \{\langle \text{ESTIMATE}, q, r_p, e_q, ts_q\rangle_q$ : $p$ received properly formed and justified
                       $\langle\langle \text{ESTIMATE}, q, r_p, e_q, ts_q\rangle_q, \text{confirms}_q\rangle_q$ from $q\}$;
          $ts \leftarrow$ largest $ts_q$: $\langle \text{ESTIMATE}, q, r_p, e_q, ts_q\rangle_q \in \text{estimates}_p$;
          if [$ts = 0$ and (for at least $k + 1$ distinct processes $q$ and common value $e$: $\langle \text{ESTIMATE}, q, r_p, e, ts\rangle_q \in \text{estimates}_p$)] then
             $es \leftarrow e$;
          else $es \leftarrow e_q$: $\langle \text{ESTIMATE}, q, r_p, e_q, ts\rangle_q \in \text{estimates}_p$;
          send $\langle\langle \text{SELECT}, p, r_p, es, ts\rangle_p, \text{estimates}_p\rangle_p$ to all;

    /* Phase 3: */
      wait until [(for $\lfloor (n + k)/2 \rfloor + 1$ distinct processes $q$ and common values $e$, $\text{select}_p$: $p$ received properly formed
                 and justified $\langle\langle \text{CONFIRM}, q, r_p, e\rangle_q, \text{select}_q\rangle_q$ from $q$) or $c_p \in \text{output}(\mathcal{D}_2)_p$];
      if [for $\lfloor (n + k)/2 \rfloor + 1$ distinct processes $q$ and common values $e$, $\text{select}_p$: $p$ received properly
         formed and justified $\langle\langle \text{CONFIRM}, q, r_p, e\rangle_q, \text{select}_p\rangle_q$] then
         $ts_p \leftarrow r_p$;
         $e_p \leftarrow e$;
         $\text{confirms}_p \leftarrow \{\langle \text{CONFIRM}, q, r_p, e\rangle_q$: $p$ received properly formed and justified $\langle\langle \text{CONFIRM}, q, r_p, e\rangle_q, \text{select}_q\rangle_q\}$;
         send $\langle\langle \text{READY}, p, r_p, e\rangle_p, \text{confirms}_p\rangle_p$ to all;
      else
         send $\langle \text{NREADY}, p, r_p\rangle_p$ to all;

  /* Task 2: */
  repeat forever
    if [($p$ received properly formed and justified $\langle\langle \text{SELECT}, c, r, e, ts\rangle_c, \text{estimates}_c\rangle_c$
        from $c \equiv (r \bmod n) + 1$) and ($p$ has not previously sent $\langle\langle \text{CONFIRM}, p, r, -\rangle_p, -\rangle_p$)] then
      $\text{select}_p \leftarrow \{\langle \text{SELECT}, c, r, e, ts\rangle_c\}$;
      send $\langle\langle \text{CONFIRM}, p, r, e\rangle_p, \text{select}_p\rangle_p$ to all;

  /* Task 3: */
  wait until [for $\lfloor (n + k)/2 \rfloor + 1$ distinct processes $q$ and a common $r$, $e$: $p$ received
              properly formed and justified $\langle\langle \text{READY}, q, r, e\rangle_q, \text{confirms}_q\rangle_q$ from $q$];
  decide($e$);

  /* Task 4: */
  repeat forever
    $\text{suspected}_p \leftarrow \mathcal{D}_{1p}$;      /* p queries its local fault detector $\mathcal{D}_{1p}$ */
    send $\langle \text{SUSPECT}, p, \text{suspected}_p\rangle_p$ to all;

  /* Task 5: */
  when $p$ receives $\langle \text{SUSPECT}, q, \text{suspected}_q\rangle_q$ from $q$
    for each $r$ in $S$
      if $r \in \text{suspected}_q$ then
         $\text{suspecting}_p[r] \leftarrow (\text{suspecting}_p[r] \cup \{q\})$;
      else
         $\text{suspecting}_p[r] \leftarrow (\text{suspecting}_p[r] - \{q\})$;
      if $|\text{suspecting}_p[r]| \geq k + 1$ then
         $\text{output}(\mathcal{D}_2)_p \leftarrow (\text{output}(\mathcal{D}_2)_p \cup \{r\})$;
      else
         $\text{output}(\mathcal{D}_2)_p \leftarrow (\text{output}(\mathcal{D}_2)_p - \{r\})$;
coend
```

FIGURE 3. An algorithm A_1 to solve consensus using any fault detector $\mathcal{D}_1$ in $\diamond\mathcal{W}(\text{Byz}, A_1)$, which is transformed by A_1 into a fault detector $\mathcal{D}_2$ in $\diamond\mathcal{S}(\text{Byz}, A_1)$.

7.1. The consensus algorithm

The consensus algorithm uses techniques from the consensus algorithm of Chandra and Toueg [3], from the reliable broadcast protocol of Bracha and Toueg [16] and from the transformation algorithm of Guerraoui and Schiper [17]. The algorithm 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 five types of messages—ESTIMATE, SELECT, CONFIRM, READY/NREADY and SUSPECT, which are described below.

The algorithm consists of five concurrent tasks. The first task consists of a sequence of rounds, each of which contains three phases.

In the first phase of task 1, each process p sends an ESTIMATE message to all.

In the second phase of task 1, the coordinator s waits to receive ESTIMATE messages from $n - k$ 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 k faulty processes, s eventually receives ESTIMATE messages from at least $n - k$ processes and thus will not wait forever in the second phase.

In the third phase of task 1, each process p waits to receive CONFIRM messages with the same value for the round[4] from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes or until it suspects the coordinator according to the fault detector $\mathcal{D}_2$, whichever occurs first. If a process p receives CONFIRM messages containing the same value from at least $\lfloor (n + k)/2 \rfloor + 1$ 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 a process p comes to suspect the coordinator before receiving the required CONFIRM messages, it sends an NREADY message to all. Note that if a correct process p never receives CONFIRM messages containing the same value from at least $\lfloor (n + k)/2 \rfloor + 1$ distinct processes, then the coordinator has detectably deviated from the algorithm. In this case, because the transformed fault detector $\mathcal{D}_2$ satisfies eventual strong Byzantine completeness for algorithm A_1, p will eventually come to suspect the coordinator. Thus, no correct process p will wait forever in the third phase.

In the second task, whenever a process p 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.

[4]The 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'.

In the third task, a process p waits to receive READY messages containing a common value for the same round from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes and then decides on that value. The third task then terminates, although execution of the other tasks continues. It is also possible to terminate the other tasks but, for simplicity of presentation, we choose not to do so.

Tasks four and five transform the fault detector $\mathcal{D}_1 \in \diamond\mathcal{W}(\text{Byz}, A_1)$ into a fault detector $\mathcal{D}_2 \in \diamond\mathcal{S}(\text{Byz}, A_1)$. To do this, a process p uses the fault detector $\mathcal{D}_1$ to maintain a variable $output(\mathcal{D}_2)_p$ that emulates the output of $\mathcal{D}_2$. In task four, a process p periodically sends the suspects list obtained from its local fault detector module $\mathcal{D}_{1p}$ to every process. In task five, a process p adds another process q to $output(\mathcal{D}_2)_p$ if and only if p has received reports from $k + 1$ processes (including itself) that suspect q according to the fault detector $\mathcal{D}_1$.

If there are no faults and no false suspicions, the consensus algorithm terminates in one round. The ESTIMATE, SELECT, CONFIRM and READY messages that are sent in such a run are shown in Figure 4. For simplicity, the messages are shown as if they were sent synchronously, but generally this is not the case. If there is a round in which the coordinator is correct and is not suspected, then consensus is reached in that round.

Once there is a round in which a correct process p can decide on a value e, then the value e is locked, using the timestamp, so that no other correct process can decide on a different value. This is proved in Appendix A, Sublemma A.6.

The consensus algorithm A_1 is designed so that it uses the fault detector only to provide the liveness property of termination, rather than the safety properties of validity, agreement and irrevocability. For example, a Byzantine coordinator might attempt to block the consensus algorithm by sending mutant SELECT messages containing a different value to different correct processes in task 1, phase 2. If this occurs, some correct processes might not be able to collect $\lfloor (n+k)/2 \rfloor + 1$ CONFIRM messages with a common value in task 1, phase 3. These correct processes will wait in task 1, phase 3 until the fault detector $\mathcal{D}_2$ satisfies the eventuality condition of eventual strong Byzantine completeness for algorithm A_1.

7.2. Forms of messages

A process that receives a message must verify the signature on the message. If the signature on the message is not valid, then the process discards the message. A process that receives a signed message must also determine that the message is properly formed and properly justified. If a message is not properly formed or is not properly justified, then the process does not accept the message for use in the consensus algorithm. This requirement is similar to the requirement of Malkhi and Reiter [6] that messages are well formed.

As described in Section 5, messages contain a statement and a justification. The entire message, containing both

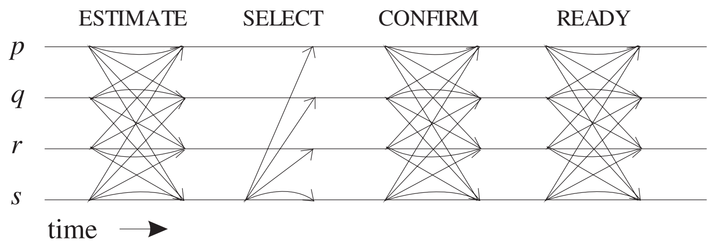

FIGURE 4. The ESTIMATE, SELECT, CONFIRM and READY messages sent during one round of the consensus algorithm A_1 for a run in which there are no faults and no false suspicions, where s is the coordinator.

signed statement and justification, is signed by the process that sends it. Both the statement and the entire message are signed so that a signed statement can be extracted from a message for use in the justification field of a subsequent message without causing the message size to grow exponentially as the number of rounds increases.

Clever use of message digests and digital signatures can allow a single signature per message to provide the functionality described above. To send a message m consisting of a message header *m.hdr*, contents *m.cont* and a justification *m.jst*, a process p computes the message digest *d(m.cont)* of the contents and the message digest *d(m.jst)* of the justification. Process p then computes the signature *m.sig* over the combination of the header and digests ⟨*m.hdr, d(m.cont), d(m.jst)*⟩. Finally, p sends the message consisting of ⟨*m.hdr, d(m.cont), d(m.jst), m.sig, m.cont, m.jst*⟩.

A process q that receives message m can verify that the signature is valid and that *d(m.cont)* and *d(m.jst)* are the digests of the contents and the justification, respectively. Process q can also include the signed statement in the justification of another message m' by including ⟨*m.hdr, d(m.cont), d(m.jst), m.sig, m.cont*⟩ in the justification field $m'.jst$. A process r that receives m' can verify that p signed the statement contained in the justification field by verifying that *d(m.cont)* is the digest of *m.cont* and that *m.sig* is a valid signature of p computed over ⟨*m.hdr, d(m.cont), d(m.jst)*⟩. The single signature on *m* thus suffices both as the signature of the entire message and as the signature of the statement. For simplicity in the description of our consensus algorithm and in Figure 3, we show two separate signatures on each message.

The *proper form* and *proper justification* of each type of message used by the consensus algorithm A_1 are now specified. The proper form of an ESTIMATE message sent by process p for round r is $\langle\langle \text{ESTIMATE}, p, r, e, ts\rangle_p$, $\mathit{confirms}\rangle_p$. The *confirms* field is the justification of the ESTIMATE statement. This field is empty if $ts = 0$ and, otherwise, contains the $\lfloor (n+k)/2 \rfloor + 1$ signed CONFIRM statements that caused p to update its estimate to e and its timestamp to ts. All of the CONFIRM statements contained in the *confirms* field must contain round ts and value e. If an ESTIMATE statement contains $r = 1$, then it must contain $ts = 0$.

The proper form of a SELECT message sent by the coordinator c for round r is $\langle\langle \text{SELECT}, c, r, e, ts\rangle_c$, $\mathit{estimates}_c\rangle_c$. The $\mathit{estimates}_c$ field is the justification of the SELECT statement. It contains $n-k$ signed ESTIMATE statements that the coordinator c received. All of the ESTIMATE statements must contain round r. The identifier of the coordinator must satisfy $c = (r \bmod n) + 1$. The values e and ts in the SELECT statement are determined by the corresponding values in the ESTIMATE statements. If a SELECT statement contains $r = 1$, then it must contain $ts = 0$.

The proper form of a CONFIRM message sent by process p for round r is $\langle\langle \text{CONFIRM}, p, r, e\rangle_p$, $\mathit{select}\rangle_p$. The *select* field is the justification of the CONFIRM statement and contains the SELECT statement that p received from the coordinator. The SELECT statement in the *select* field must contain round r and value e.

The proper form of a READY message sent by process p for round r is $\langle\langle \text{READY}, p, r, e\rangle_p$, $\mathit{confirms}\rangle_p$. The *confirms* field is the justification of the READY statement. It contains $\lfloor (n+k)/2 \rfloor + 1$ signed CONFIRM statements that p received. All of the CONFIRM statements contained in the *confirms* field must contain round r and value e. The proper form of an NREADY message sent by process p for round r is $\langle \text{NREADY}, p, r\rangle_p$. The justification field of an NREADY message is empty.

The proper form of a SUSPECT message sent by process p is $\langle \text{SUSPECT}, p, \mathit{suspected}\rangle_p$. The *suspected* field contains the suspects list obtained from the local fault detector module $\mathcal{D}1_p$. The justification field of a SUSPECT message is empty.

In Appendix A we establish the correctness of the consensus algorithm A_1 of Figure 3. We prove the following theorem by showing that it satisfies the termination, validity, agreement and irrevocability properties of consensus.

THEOREM 7.1. *The consensus algorithm A_1 of Figure 3 solves the consensus problem using any fault detector in $\Diamond\mathcal{W}(\text{Byz}, A_1)$ in an asynchronous distributed system with n processes, where the maximum number k of Byzantine faults satisfies $k \le \lfloor (n-1)/3 \rfloor$.*

8. IMPLEMENTING BYZANTINE FAULT DETECTORS

We now describe how Byzantine fault detectors can be implemented. The local fault detector module at process p monitors messages that are sent and received by the algorithm A and provides an output consisting of a list output_p of processes that p currently suspects of having detectably deviated from algorithm A. Detectable deviation from algorithm A is defined in terms of omission faults and commission faults, as described in Section 4.

The fault detector makes use of timeouts to detect omission faults. In a system subject to crash faults, timeouts can be used in the following way. A process p sends an 'are you alive?' message to another process and waits for the response. However, this will not work in a system subject to Byzantine faults, because a Byzantine process can faithfully respond to 'are you alive?' messages while refusing to participate in the algorithm A.

Thus, we make use of timeouts that are set for messages sent by algorithm A. When a correct process p *sets* a timeout, an interval begins during which p expects to receive a certain message required by algorithm A. 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 message by the end of the interval. The timeout is canceled if p receives the expected message before the end of the interval.

In an asynchronous system, the expiration of a timeout does not necessarily indicate that a fault has occurred; a process may be slow or message transmission may be slow. Therefore, it may be desirable to make adjustments to the timeout interval such that if a timeout expires and then the expected message is subsequently received, the timeout interval is increased. An implementation of a fault detector based on this principle has been given by Chandra and Toueg [3]. We also make use of this technique.

Although we desire to capture as much information about faults as possible, in a Byzantine environment we are unable to ensure that correct processes will detect the omission of a single point-to-point message. Thus, we must mask a fault in which a Byzantine process sends a message to some correct processes but not to others. We must ensure that, if any correct process receives a message, then all correct processes eventually receive the message. There are several ways to achieve this. The approach we take is to require that, when a correct process receives a message, it must immediately relay the message to all processes.

As described in Section 4, there are two types of commission faults: (1) a process sends a message that does not conform to the algorithm; and (2) a process sends two or more mutant messages. In an asynchronous system, it is impossible to know with certainty whether an omission fault has occurred, a process is simply slow, or a message has been delayed. However, if a correct process receives an improperly formed or justified message, or two mutant messages, then that process knows with certainty that a commission fault has occurred (assuming that the signature scheme is unforgeable). For example, if p receives properly signed mutant messages m and m' from q, then p has proof that q has deviated from the algorithm. In such a case, p knows that q is Byzantine and that it should permanently suspect q. To implement this, the fault detector at p maintains a list $byzset_p$ of processes that p knows to be Byzantine.

A general method for implementing a Byzantine fault detector is then as follows. Set timeouts for all expected messages; if a timeout expires for a message from process q, conditionally add q to the suspects list; if the message later arrives, then q may be removed. Additionally, check each message from q to determine that it is properly formed and justified and is not a mutant of another message from q; if these criteria are not met, permanently add p to the suspects list.

8.1. A fault detector for a general algorithm

We now present an algorithm that implements, in the $\mathcal{M}_3$ model, a fault detector $\mathcal{D} \in \Diamond\mathcal{P}(\text{Byz}, A)$ for a general algorithm A that uses a fault detector. The assumptions that we make regarding algorithm A are described in Section 5.

The fault detector algorithm executed by a correct process p consists of three concurrent tasks, as shown in Figure 5. In the first task, when p expects a required message m, p sets a timeout for that message. When we say that p expects a required message m, where $m.sender = q$, we mean that p expects a message that is signed by q and that has a particular message header.

In the second task, when a timeout expires for a required message m, where $q = m.sender$, p adds q to its list $output_p$ containing the identifiers of the processes it suspects. Process p retains the information regarding the specific message header it expected in the list $expecting_p[q]$.

In the third task, when p receives a message m signed by q for the first time, where $q = m.sender$ and $q \neq p$, p sends m in its original form to all processes. Next, p checks whether message m is properly formed and properly justified or whether m is a mutant of a message m' signed by q that p has already received. If m is not properly formed or is not properly justified or is a mutant, then p adds q to its suspects list $output_p$ and also to its list $byzset_p$ to indicate that q should never subsequently be removed from $output_p$.

If message m is properly formed and properly justified and is not a mutant, then p cancels the timeout for message m if it has not already expired. If m is in $expecting_p[q]$ (where $q = m.sender$), then a premature timeout has occurred and so p removes m from $expecting_p[q]$ and increases the timeout interval for q. If $expecting_p[q]$ is now empty and q is not contained in $byzset_p$, then p removes q from $output_p$.

It is possible that different timeout values are appropriate for the different types of messages sent by the algorithm. In this case, the algorithm given in Figure 5 can be modified to include a separate timeout interval for each message type from each process.

In Appendix B we establish the correctness of the algorithm given in Figure 5, assuming a partially synchronous system S that conforms to $\mathcal{M}_3$. That is, we prove the following theorem, assuming that for every run of the algorithm A in S there is a GST after which some bounds on processing and message transmission times hold, although the values of GST and these bounds are not known *a priori*.

THEOREM 8.1. *The algorithm given in Figure 5 implements an eventually perfect Byzantine fault detector* $\mathcal{D} \in \Diamond\mathcal{P}(\text{Byz}, A)$ *for a general algorithm A that uses a fault detector.*

```
/* Each process p executes the following */
/* Initialization */
$output_p \leftarrow \emptyset$;                                    /* Set of processes that p suspects */
$byzset_p \leftarrow \emptyset$;                                    /* Set of processes that p 'knows' to be Byzantine */
for each $q \in S$
  $\Delta_p(q) \leftarrow$default value;                            /* Duration of p's timeout interval for q */
  $expecting_p[q] \leftarrow \emptyset$;                            /* Required messages p expected from q when p's timeout expired */
cobegin                                                             /* Three concurrent tasks */
  /* Task 1: */
  when [$p$ expects a required message $m$, where $q = m$.sender]
    set timeout of duration $\Delta_p(q)$ for message $m$;
  /* Task 2: */
  when [$p$'s timeout expires for a required message $m$, where $q = m$.sender]   /* Omission fault */
    $output_p \leftarrow output_p \cup \{q\}$;
    $expecting_p[q] \leftarrow expecting_p[q] \cup \{m\}$;
  /* Task 3: */
  when [$p$ receives a properly signed required message $m$ for the first time, where $q = m$.sender]
    if [$m$ is not properly formed or $m$ is not properly justified or
          $m$ is a mutant of a previous message $m'$] then           /* Commission fault */
        $output_p \leftarrow output_p \cup \{q\}$;
        $byzset_p \leftarrow byzset_p \cup \{q\}$;
    else
      cancel timeout for message $m$;
      if [$m \in expecting_p[q]$] then                              /* Premature timeout has occurred */
        $expecting_p[q] \leftarrow expecting_p[q] - \{m\}$;
        $\Delta_p(q) \leftarrow \Delta_p(q) + 1$;                   /* Increase the duration of p's timeout interval for q */
        if [$expecting_p[q] = \emptyset$ and $q \notin byzset_p$] then
          $output_p \leftarrow output_p - \{q\}$;
    if [$q \neq p$] then
      send $m$ to all;                                              /* Relay message m */
coend
```

FIGURE 5. An algorithm for a fault detector $\mathcal{D} \in \Diamond\mathcal{P}(\text{Byz}, A)$ in a model of partial synchrony and unforgeable signatures, where A is a general algorithm that uses a fault detector.

8.2. A fault detector for the consensus algorithm

We now present an algorithm that implements, in the $\mathcal{M}_3$ model, a fault detector $\mathcal{D} \in \Diamond\mathcal{P}(\text{Byz}, A_1)$ for the consensus algorithm A_1 of Figure 3. This algorithm is a specific instance of the general algorithm given in Figure 5 and is provided simply to give a concrete example of the fault detector implementation.

The specific fault detector algorithm executed by a correct process p is shown in Figure 6 and consists of three concurrent tasks, which correspond with the three tasks of the algorithm of Figure 5. For simplicity, we do not show the justification field of messages and sometimes we show multiple fields containing an arbitrary value as a single field. The variable TYPE is used to indicate a message type chosen from the values ESTIMATE, SELECT, CONFIRM and READY/NREADY. The first task involves the setting of timeouts, the second task is executed when a timeout expires and the third task is executed when a message is received for the first time.

The first task is divided into four subtasks. In the first subtask, when p enables round r, p sets a timeout for receiving an ESTIMATE message for round r from each process. In the second subtask, when p receives an ESTIMATE message for round r from $n - k$ processes, p sets a timeout for receiving a SELECT message from the coordinator of round r. In the third subtask, when p receives a SELECT message from the coordinator of round r, p sets a timeout for receiving a CONFIRM message for round r from each process. In the fourth subtask, when p receives CONFIRM messages containing a common round and value from $\lfloor (n+k)/2 \rfloor + 1$ processes, p sets a timeout for receiving a READY or NREADY message for that round from each process.

In the second task, when a timeout expires for a message m from process q, p adds q to the suspects list $output_p$ and adds m to the list $expected_p[q]$ of messages that p expected but did not receive from q.

The third task is executed whenever p first receives a message m sent by q. The reception can occur in two ways: either p receives m directly from q, or p receives, from another process s, a message that q signed and sent to s and that s relayed to p. In either case, if m is not properly formed or is not properly justified according to the definitions in Section 7.2, or is a mutant of a message m' that p previously received, p adds q to its suspects list $output_p$ and also to its list $byzset_p$ to indicate that q should never subsequently be removed from $output_p$.

If m is properly formed and properly justified and is not a mutant of a previous message, then p cancels the timeout for m if it has not yet expired. However, if m is in $expected_p[q]$ (indicating that p has prematurely timed out on message m from q), then p removes m from $expected_p[q]$, increases the timeout interval for messages

```
/* Each process p executes the following */
/* Initialization */
output_p ← ∅;                                   /* The set of processes suspected by p */
byzset_p ← ∅;                                   /* The set of processes 'known' by p to be Byzantine */
for each $q \in S$
  $\Delta_p(q)$ ← default value;                  /* The duration of p's timeout interval for q */
  expecting_p[q] ← ∅;                           /* The set of messages p expected from q when p's timeout expired */
cobegin                                          /* Three concurrent tasks */
  /* Task 1: */
  /* Subtask 1-A: */
  when [p enables round r]
    for each $q \in S$
      set timeout of duration $\Delta_p(q)$ for receiving $m = \langle \text{ESTIMATE}, q, r, - \rangle_q$;
  /* Subtask 1-B: */
  when [for $n - k$ distinct processes $q$: $p$ received properly formed and justified
        $\langle \text{ESTIMATE}, q, r, - \rangle_q$]
    set a timeout of duration $\Delta_p(c)$ for receiving $m = \langle \text{SELECT}, c, r, - \rangle_c$: $c \equiv (r \bmod n) + 1$;
  /* Subtask 1-C: */
  when [for $\lfloor (n + k)/2 \rfloor + 1$ distinct processes $q$ and common values $e, r$: $p$ received properly
        formed and justified $\langle \text{CONFIRM}, q, r, e \rangle_q$]
    for each $s \in S$
      set a timeout of duration $\Delta_p(s)$ for receiving $m = \langle \text{READY}, s, r, e \rangle_s$ or $m = \langle \text{NREADY}, s, r \rangle_s$;
  /* Subtask 1-D: */
  when [p received properly formed and justified $\langle \text{SELECT}, c, r, e, - \rangle_c$: $c \equiv (r \bmod n) + 1$]
    for each $q \in S$
      set a timeout of duration $\Delta_p(q)$ for receiving $m = \langle \text{CONFIRM}, q, r, e \rangle_q$;
  /* Task 2: */
  when [p's timeout expires for receiving $m = \langle \text{TYPE}, q, r, - \rangle_q$]      /* Omission fault */
    output_p ← (output_p ∪ {q});
    expecting_p[q] ← (expecting_p[q] ∪ {m});
  /* Task 3: */
  when [p receives $m = \langle \text{TYPE}, q, r, - \rangle_q$ for the first time]
    if [m is not properly formed or m is not properly justified or
        m is a mutant of a previous message m′] then                  /* Commission fault */
      output_p ← (output_p ∪ {q});
      byzset_p ← (byzset_p ∪ {q});
    else
      cancel timeout for message m;
      if [m ∈ expecting_p[q]] then                                     /* Premature timeout has occurred */
        expecting_p[q] ← (expecting_p[q] − {m});
        $\Delta_p(q) \leftarrow \Delta_p(q) + 1$;                      /* Increase the duration of p's timeout interval for q */
        if [expecting_p[q] = ∅ and $q \notin$ byzset_p] then
          output_p ← (output_p − {q});
    if [$q \neq p$] then
      send m to all;                                                   /* Relay message m */
coend
```

FIGURE 6. An implementation of a fault detector $\mathcal{D} \in \Diamond\mathcal{P}(\text{Byz}, A_1)$, where A_1 is the consensus algorithm of Figure 3, in a model of partial synchrony.

from q, and checks to see if q should be removed from its suspects list *output*$_p$. If q is not in *byzset*$_p$ and there is no remaining message in *expected*$_p[q]$, then p removes q from *output*$_p$. Every time p executes the third task, p relays the message m if p is not the process that originally sent the message.

In Appendix B we establish the correctness of the algorithm given in Figure 6, assuming a partially synchronous system S that conforms to $\mathcal{M}_3$. That is, we prove the following theorem, assuming that for every run of the consensus algorithm A_1 in S there is a GST after which some bounds on processing and message transmission times hold, although the values of GST and these bounds are not known *a priori*.

THEOREM 8.2. *The algorithm given in Figure 6 implements an eventually perfect Byzantine fault detector* $\mathcal{D} \in \Diamond\mathcal{P}(\text{Byz}, A_1)$*, where* A_1 *is the algorithm given in Figure 3.*

8.2.1. *Optimization*

We observe that, in the consensus algorithm of Chandra and Toueg [3] that solves consensus using $\Diamond\mathcal{S}$, a process relies only on the fault detector for information regarding the coordinator. Thus, in any particular round, detecting faulty processes other than the coordinator is not required, while detecting a faulty coordinator is essential before correct processes can move on to the next round. The reason is that the consensus algorithm masks some faults, e.g. the

```
/* Each process p executes the following */
/* Initialization */
output_p ← ∅;                                        /* The set of processes suspected by p */
timeouts_p ← ∅;                                      /* The set of round numbers for which timeouts have expired */
for each $q \in S$
  $\Delta_p(q)$ ← default value;                     /* The duration of p's timeout interval for q */
cobegin                                              /* Three concurrent tasks */
  /* Task 1: */
  when [$p$ sends $\langle$ESTIMATE, $p, r, -\rangle_p$ to $c \equiv (r \bmod n) + 1$]
    set timeout of duration $\Delta_p(c)$ for round $r$;
  /* Task 2: */                                      /* Omission fault */
  when [$p$'s timeout for round $r$ expires]
    timeouts_p ← (timeouts_p ∪ {$r$});
    output_p ← (output_p ∪ {$c : c \equiv (r \bmod n) + 1$});
  /* Task 3: */
  when [for $\lfloor (n+k)/2 \rfloor + 1$ distinct processes $q$ and common values $e, r$: $p$ received
          properly formed and justified $\langle$CONFIRM, $q, r, e\rangle_q$ from $q$]
    if [$r \notin timeouts_p$] then
      cancel timeout for round $r$;
    else                                             /* Premature timeout has occurred for round r */
      $c \leftarrow (r \bmod n) + 1$;                /* Determine the coordinator of round r */
      $\Delta_p(c) \leftarrow \Delta_p(c) + 1$;      /* Increase the duration of p's timeout interval for c */
      timeouts_p ← (timeouts_p − {$r$});
      if [for all $r' \in$ timeouts_p : $(r' \bmod n) + 1 \not\equiv c$] then
        output_p ← (output_p − {$c$});
coend
```

FIGURE 7. An optimized implementation of a Byzantine fault detector that can be used in conjunction with the consensus algorithm A_1 of Figure 3, in a model of partial synchrony.

coordinator is required to wait only for responses from $\lceil (n + 1)/2 \rceil$ processes.

Similarly, for the consensus algorithm A_1 of Figure 3, detecting faulty processes other than the coordinator is not required. Thus, the completeness property of the fault detector matters only for the coordinator. This observation is particularly significant in the case of Byzantine faults, because detectable deviation from the algorithm might not be permanent. For example, a Byzantine process can refuse to send messages during a round for which it is not the coordinator, but then behave according to the algorithm while it is the coordinator.

As noted previously, the fault detector works together with the algorithm to provide information about faults. A tradeoff exists between detecting as many faults as possible and making the algorithms as efficient as possible. Because some faults are masked by the consensus algorithm A_1, we can optimize both the consensus algorithm and the fault detector algorithm so that fewer messages are sent. In return, we give up some knowledge about faults.

The consensus algorithm A_1 of Figure 3 can be optimized by making the following modifications. In phase 3 of task 1, a process p that suspects the coordinator proceeds to the next round without sending an NREADY message. Furthermore, when the consensus algorithm is used in conjunction with a fault detector that satisfies eventual strong Byzantine completeness, tasks four and five of the consensus algorithm, which transform a fault detector satisfying eventual weak Byzantine $(k + 1)$-completeness into a fault detector satisfying eventual strong Byzantine completeness, can be eliminated.

The fault detector of Figure 6 can be optimized by ensuring that, for every correct process p executing round r for which c is the coordinator, at least one of the following holds:

- process p collects, from at least $\lfloor (n + k)/2 \rfloor + 1$ processes, CONFIRM messages for round r with a common value e;
- process p eventually comes to suspect the coordinator c.

The optimized fault detector algorithm is shown in Figure 7. The algorithm executed by a correct process p consists of three concurrent tasks. In the first task, when p sends an ESTIMATE message to the coordinator of round r, p sets a timeout for round r.

In the second task, when a timeout expires for round r, p adds the coordinator of round r to its list $output_p$, containing the identifiers of the processes it suspects and adds r 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 from $\lfloor (n+k)/2 \rfloor + 1$ distinct processes, p cancels the timeout for round r if it has not yet expired. However, if the timeout has already expired (indicating that p has prematurely timed out for round r), then p determines the coordinator c for round r, increases the timeout interval for c, and removes r from $timeouts_p$. If there is no remaining round number in $timeouts_p$ for which c was the coordinator, then p removes c from $output_p$.

A subtle point arises. In task 3, p waits to receive CONFIRM messages from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes, but if the timeout expires, it is the coordinator c that p will suspect, even if p received a SELECT message from c and even if p received a CONFIRM message from c. The reason is the following. If the coordinator c is Byzantine, it may send a SELECT message to a proper subset of correct processes rather than to all of them, or may send a SELECT message m to some of the correct processes and send a mutant SELECT message m' to other correct processes. In this case, a correct process p may not be able to collect CONFIRM messages containing a common value from $\lfloor (n+k)/2 \rfloor + 1$ processes. Therefore, p will wait in phase 3 of task 1 of the consensus algorithm until p comes to suspect the coordinator c.

If p's timeout expires before it collects the CONFIRM messages, then in task 3 of the fault detector p will come to suspect c and will not wait forever in phase 3 of task 1. On the other hand, if the coordinator is correct, then each correct process p will eventually receive the same SELECT message and will eventually send a CONFIRM message. Because there are at least $\lfloor (n+k)/2 \rfloor + 1$ correct processes, each correct process p will eventually receive CONFIRM messages from $\lfloor (n+k)/2 \rfloor + 1$ distinct processes containing a common value.

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 message transmission is slow, then p may incorrectly come to suspect the coordinator c. However, such a mistake by the fault detector is allowed (before the GST) according to the eventual strong accuracy property. Eventually, when p receives the CONFIRM message sent by q, it will remove c from its list of suspects $output_p$.

9. COMPLEXITY

We now consider the cost of our consensus algorithm in terms of the latency degree and the expected number of messages.

9.1. Latency degree

Schiper [18] introduced the notion of *latency degree* for measuring the cost of a distributed algorithm. Each event and each message has a timestamp associated with it; these timestamps are not related to those of Figure 3. Schiper defines the latency degree by considering logical clocks and the following rules.

- A *send* event and a *local* event at a process p do not modify p's logical clock value.
- If *t(send(m))* is the timestamp of the *send(m)* event and *t(m)* is the timestamp of a message m, then $t(m) \overset{\text{def}}{=}$ *t(send(m))*+1.
- If *t(receive(m))* is the timestamp of the *receive(m)* event at a process p, then *t(receive(m))* is the maximum of $t(m)$ 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. Schiper [18] has noted that the algorithm of Chandra and Toueg [3] 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.

Given this notion of latency degree and assuming the use of digital signatures, we have the following theorem. The proof can be found in Appendix C. Figure 4 serves to illustrate the latency degree for the consensus algorithm.

THEOREM 9.1. *The consensus algorithm A_1 given in Figure 3 has a latency degree of 4.*

In contrast, the consensus algorithm of Malkhi and Reiter [6] has a latency degree of 9 if the reliable broadcast service is implemented using the protocol of Bracha and Toueg [16] or Reiter [15] and a latency degree of 6 if the reliable broadcast service is implemented using the protocol of Malkhi and Reiter [19]. Malkhi and Reiter assume the existence of a reliable authenticated communication channel between each pair of servers. Two of the implementations that Malkhi and Reiter cite for the reliable broadcast service employ digital signatures [15, 19], while the third [16] does not.

9.2. Expected number of messages

In determining the expected number of messages required by the consensus algorithm A_1 of Figure 3, we do not consider messages sent in task four of the algorithm A_1 for the following reasons. The frequency with which the messages are sent in task four is not determined by algorithm A_1. Moreover, when algorithm A_1 is used in conjunction with a fault detector (such as the fault detector of Figure 6) that satisfies eventual strong Byzantine completeness for algorithm A_1, tasks four and five of the consensus algorithm, which transform a fault detector satisfying eventual weak Byzantine $(k+1)$-completeness for algorithm A_1 into a fault detector satisfying eventual strong Byzantine completeness for algorithm A_1, do not need to be executed.

To estimate the number of messages for the consensus algorithm, we let k denote the resilience to Byzantine processes and let b denote the actual number of Byzantine processes. In Lemma C.1, we show that the number of broadcast messages required by the consensus algorithm A_1 for one round is equal to $3n+1$.

We now consider whether or not a consensus decision is reached in a round, depending on whether the coordinator is correct or Byzantine and whether the coordinator is or is not suspected.

If the first coordinator is correct and is not suspected by any correct process, the algorithm terminates in one round. In the absence of false suspicions, the maximum number of rounds for a process to decide is $b+1$ rounds. This occurs when the coordinators of the first b rounds are Byzantine and no decision is reached in any of those rounds. Thus, the coordinator of the $(b+1)st$ round is correct and a decision is reached in that round.

If the coordinator for a round is correct but is falsely suspected by some of the processes, the algorithm may or may not decide in that round. A correct process that suspects the coordinator does not send a READY message in phase 3 of task 1 of the consensus algorithm A_1. For a correct process to decide in task 3 of A_1, it must have received, from $\lfloor (n+k)/2 \rfloor + 1$ processes, READY messages containing a common value and round number. We let ρ denote the probability that at least $\lfloor (n-k)/2 \rfloor$ processes falsely suspect a particular correct process that is acting as coordinator, so that a decision is not reached in the round.

If the coordinator for a round is Byzantine but sends the proper messages and enough processes do not suspect the coordinator, then it is possible for a decision to be reached in the round.

If the coordinator for a round is Byzantine and enough processes suspect the coordinator, then a decision cannot be reached in that round. We let σ denote the probability that at least $\lfloor (n-k)/2 \rfloor$ processes suspect a particular Byzantine process that is acting as coordinator, so that a decision is not reached in the round.

For the consensus algorithm A_1 of Figure 3, we have the following theorem, which we prove in Appendix C.

THEOREM 9.2. *The expected number of broadcast messages needed by the consensus algorithm A_1 given in Figure 3 to decide on a value in a system with n processes of which b are Byzantine is*

$$(3n+1)\left(n\frac{\rho^{n-b}\sigma^b}{1-\rho^{n-b}\sigma^b} + \sum_{i=0}^{n-1}\sum_{j=0}^{i}\frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^j\right).$$

10. CONCLUSION

We have investigated the use of fault detectors for solving the consensus problem in asynchronous distributed systems that are subject to Byzantine faults. We have provided a categorization of Byzantine faults, and have identified two new completeness properties, eventual strong Byzantine completeness for algorithm A and eventual weak Byzantine $(k+1)$-completeness for algorithm A. Using these completeness properties, we have introduced four new classes of unreliable Byzantine fault detectors. We have presented an algorithm that solves consensus in an asynchronous distributed system of n processes with at most $\lfloor (n-1)/3 \rfloor$ Byzantine faults, using a Byzantine fault detector.

Our approach has been to attempt to capture as much information about Byzantine faults as possible. In particular, this is necessary if Byzantine fault detectors are to be used to solve other consensus-related problems, such as the group membership problem; this is an area for future investigation.

Another area for future work would be to develop an API that would allow the application to interact with the fault detector. The algorithm A that uses the fault detector could send information that is specifically for the purpose of detecting Byzantine faults but may not be useful to the application. Such an interface must be carefully designed, as the application could pass Byzantine information to its local fault detector. Thus, this information must be sent to all processes and must be subject to mechanisms to detect mutant messages, etc. An example of this idea would be to require the application to periodically send a message containing a checksum taken over previous messages.

ACKNOWLEDGEMENTS

The authors thank the anonymous referees, whose insightful comments have greatly improved this paper.

This research was sponsored by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Rome, USAF, under contract number F30602-95-1-0048. Kim Kihlstrom was also supported by a University of California Doctoral Scholar Fellowship whilst a Ph.D. student.

REFERENCES

[1] Kihlstrom, K. P., Moser, L. E. and Melliar-Smith, P. M. (1997) Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proc. Int. Conf. on Principles of Distributed Systems*, Chantilly, France, December 10–12, pp. 61–75. Editions Hermes, Paris.

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

[3] Chandra, T. and Toueg, S. (1996) Unreliable failure detectors for reliable distributed systems. *J. ACM*, **43**, 225–267.

[4] Chandra, T., Hadzilacos, V. and Toueg, S. (1996) The weakest failure detector for solving consensus. *J. ACM*, **43**, 685–722.

[5] Dwork, C., Lynch, N. and Stockmeyer, L. (1988) Consensus in the presence of partial synchrony. *J. ACM*, **35**, 288–323.

[6] Malkhi, D. and Reiter, M. (1997) Unreliable intrusion detection in distributed computations. In *Proc. 10th Computer Security Foundations Workshop*, Rockport, MA, June 10–12, pp. 116–124. IEEE Computer Society Press, Los Alamitos, CA.

[7] Doudou, A. and Schiper, A. (1998) Muteness detectors for consensus with Byzantine processes. In *Proc. 17th Ann. ACM Symp. on Principles of Distributed Computing*, Puerto Vallarta, Mexico, June 28–July 2, p. 315. ACM, New York. (Brief announcement.)

[8] Doudou, A., Garbinato, B., Guerraoui, R. and Schiper, A. (1999) Muteness failure detectors: specification and implementation. In *Proc. 3rd Eur. Dependable Computing Conf.*, Prague, Czech Republic, September 15–17. *Lecture Notes in Computer Science*, **1667**, 71–87. Springer, Berlin.

[9] Alvisi, L., Malkhi, D., Pierce, E. and Reiter, M. K. (2001) Fault detection for Byzantine quorum systems. *IEEE Trans. Parallel Distrib. Syst.*, **12**, 996–1007.

[10] Rivest, R. L., Shamir, A. and Adleman, L. (1978) A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, **21**, 120–126.

[11] Rivest, R. L. (1992) *The MD5 Message Digest Algorithm*. RFC 1321, Network Working Group, MIT Laboratory for Computer Science and RSA Data Security, Inc., Cambridge, MA. Available at: http://www.faqs.org/rfcs/rfc1321.html.

[12] Bracha, G. (1987) Asynchronous Byzantine agreement protocols. *Inform. Computat.*, **75**, 130–143.

[13] Coan, B. A. (1988) A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Trans. Comput.*, **37**, 1541–1553.

[14] Neiger, G. and Toueg, S. (1990) Automatically increasing the fault-tolerance of distributed algorithms. *J. Algorithms*, **11**, 374–419.

[15] Reiter, M. K. (1994) Secure agreement protocols: reliable and atomic group multicast in Rampart. In *Proc. 2nd ACM Conf. on Computer and Communications Security*, Fairfax, VA, November 2–4, pp. 68–80. ACM, New York.

[16] Bracha, G. and Toueg, S. (1985) Asynchronous consensus and broadcast protocols. *J. ACM*, **32**, 824–840.

[17] Guerraoui, R. and Schiper, A. (1996) 'Γ-accurate' failure detectors. In *Proc. 10th Int. Workshop on Distributed Algorithms*, Bologna, Italy, October 9–11. *Lecture Notes in Computer Science*, **1151**, 269–286. Springer, Berlin.

[18] Schiper, A. (1997) Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, **10**, 149–157.

[19] Malkhi, D. and Reiter, M. (1997) A high-throughput secure reliable multicast protocol. *J. Comput. Security*, **5**, 113–127.

APPENDIX A

To establish that the consensus algorithm A_1 of Figure 3 is correct, we prove Theorem 7.1 below using Propositions A.1 and A.2, which establish that the algorithm A_1 transforms eventual weak Byzantine $(k+1)$-completeness into eventual strong Byzantine completeness while preserving eventual weak accuracy. The proof of Proposition A.1 follows from Lemmas A.1 and A.2. The proof of Proposition A.2 follows from Lemmas A.3, A.4, A.5 and A.6, which establish that the algorithm A_1 satisfies the termination, validity, agreement and irrevocability properties of consensus.

LEMMA A.1. (Transforming eventual Byzantine completeness) *Let $\mathcal{D}_1$ be a fault detector that satisfies eventual weak Byzantine $(k+1)$-completeness for algorithm A_1. Then algorithm A_1 transforms $\mathcal{D}_1$ into a fault detector $\mathcal{D}_2$ that satisfies eventual strong Byzantine completeness for algorithm A_1, provided that $n > 2k$.*

Proof. Let r be any process that detectably deviates from algorithm A_1. Because $\mathcal{D}_1$ satisfies eventual weak Byzantine $(k+1)$-completeness for algorithm A_1, there is a time after which $k+1$ correct processes $q_1, q_2, \ldots, q_{k+1}$ permanently suspect r. Each q_i sends $\langle \text{SUSPECT}, q_i, \mathit{suspected}_{q_i} \rangle_{q_i}$, where $r \in \mathit{suspected}_{q_i}$, to all processes. Every correct process s eventually receives those messages and puts r into output_s forever. □

LEMMA A.2. (Preserving eventual weak accuracy) *Let $\mathcal{D}_1$ be a fault detector that satisfies eventual weak accuracy. Then algorithm A_1 transforms $\mathcal{D}_1$ into a fault detector $\mathcal{D}_2$ that satisfies eventual weak accuracy, provided that $n > k$.*

Proof. Because $\mathcal{D}_1$ satisfies eventual weak accuracy, there is a time after which every correct process p does not suspect some correct process r and sends $\langle \text{SUSPECT}, p, \mathit{suspected}_p \rangle_p$, where $r \notin \mathit{suspected}_p$, to all processes. Because there are at most k Byzantine processes and at least $n-k$ correct processes, eventually every correct process p receives such a message from at least $n-k$ processes and, thus, $|\mathit{suspecting}_p[r]| \leq k$. Consequently, there is a time after which p removes r from output_p forever. □

PROPOSITION A.1. *The algorithm A_1 transforms eventual weak Byzantine $(k+1)$-completeness for algorithm A_1 into eventual strong Byzantine completeness for algorithm A_1 while preserving eventual weak accuracy in an asynchronous distributed system of n processes, provided that $n > 2k$.*

Proof. The proof follows from Lemmas A.1 and A.2. □

LEMMA A.3. (Termination) *Every correct process eventually decides on some value.*

Proof. Because the unreliable Byzantine fault detector $\mathcal{D}_2$ employed in task 1 by the consensus algorithm A_1 of Figure 3 is in $\Diamond\mathcal{S}(\text{Byz}, A_1)$, $\mathcal{D}_2$ satisfies eventual strong Byzantine completeness for algorithm A_1 as defined in Section 6 and eventual weak accuracy as defined in Section 2.

If the coordinator of round r detectably deviates from algorithm A_1, so that in task 1, phase 3 a correct process p cannot obtain $\lfloor (n+k)/2 \rfloor + 1$ properly formed and justified CONFIRM messages for round r containing the same value then, by eventual strong Byzantine completeness for algorithm A_1, there is a time after which p permanently suspects the coordinator, exits task 1, phase 3 and proceeds to the next round and another coordinator. In the next round with the new coordinator, process p will likewise wait to obtain $\lfloor (n+k)/2 \rfloor + 1$ properly formed and justified CONFIRM messages for that round. If p's fault detector suspects the new coordinator before p has obtained those messages, then it moves on to the next round and another coordinator and so on. By eventual weak accuracy, there is a time t after which some correct process c is never suspected by any correct process.

We now argue by contradiction. Suppose that some correct process has not yet decided on a value at the first time t', $t' \geq t$, that any correct process begins a round r' for which c is the coordinator.

In task 1, phase 2 of round r', the coordinator c selects a value e' determined by the set of ESTIMATE messages that it has received and sends a SELECT message containing that value. Because c is a correct process, each correct process eventually receives c's SELECT message. In task 2, each correct process sends a CONFIRM message containing value e' for round r'. These CONFIRM messages are eventually received by each correct process.

Because each correct process p does not suspect c after time t, p waits to receive $\lfloor (n+k)/2 \rfloor + 1$ CONFIRM messages containing value e' for round r' in task 1, phase 3 of round r'. It then sends a READY message containing the value e'. In task 3, the correct process p receives $\lfloor (n+k)/2 \rfloor + 1$ READY messages containing the same value e' and round r'

and, thus, decides on e'. This contradicts the above supposition and demonstrates the lemma. □

We now prove Sublemma A.1, which is used to prove the validity property of consensus in Lemma A.4 below.

SUBLEMMA A.1. *If all correct processes have the same input value, then:*

(1) *the value e in a properly formed and justified* ESTIMATE *message for round r with timestamp $ts > 0$ is the initial value of the correct processes;*
(2) *the value e in a properly formed and justified* SELECT *message for round r is the initial value of the correct processes;*
(3) *the value e in a properly formed and justified* CONFIRM *message for round r is the initial value of the correct processes;*
(4) *the value e in a properly formed and justified* READY *message for round r is the initial value of the correct processes.*

Proof. The proof is by induction on r. In the base case, $r = 1$ and, thus, the timestamp $ts = 0$. Statement (1) is vacuously satisfied for $r = 1$.

To establish statement (2) for $r = 1$, we first note that in task 1, phase 1 of round 1 each correct process p sends its initial value in its ESTIMATE message. Because all correct processes have the same input value, the value e that the coordinator includes in its SELECT message is the common value contained in $k + 1$ of the $n - k$ ESTIMATE messages that it received for round 1. Because at least one of those ESTIMATE messages is from a correct process, e is the initial value of the correct processes.

Statement (3) for $r = 1$ follows from statement (2) for $r = 1$ because correct processes confirm only properly formed and justified SELECT messages. Similarly, statement (4) for $r = 1$ follows from statement (3) for $r = 1$.

To establish the inductive step, we assume that the four statements hold for all r', $1 \leq r' \leq r$ and prove that they hold for $r + 1$. We demonstrate statement (1) for round $r + 1$ by first noting that a properly formed and justified ESTIMATE message for round $r+1$ with $ts > 0$ contains $\lfloor (n+k)/2 \rfloor + 1$ properly formed CONFIRM statements for round r', where $r' = ts \leq r$. At least $\lfloor (n+k)/2 \rfloor + 1 - k = \lfloor (n-k)/2 \rfloor + 1$ of those CONFIRM statements were sent by correct processes and, thus, contain the initial value of the correct processes by statement (3) for round r.

To demonstrate statement (2) for round $r+1$, we first note that if $ts = 0$ in the SELECT message, then the argument is identical to that for the base case. If $ts > 0$, then the value e in the SELECT message for round $r + 1$ is the value in a properly formed and justified ESTIMATE message for round $r+1$. The value in that ESTIMATE message is the initial value of the correct processes by statement (1) for round $r + 1$.

Statement (3) for round $r + 1$ follows from statement (2) for round $r + 1$ because correct processes confirm only properly formed and justified SELECT messages. Similarly, statement (4) for round $r + 1$ follows from statement (3) for round $r + 1$. □

LEMMA A.4. (Validity) *If all correct processes have the same input value, then every correct process that decides must decide on that input value.*

Proof. If a correct process p decides on value e, then p has received properly formed and justified READY messages containing value e from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes. Such a properly formed and justified message might be sent by a Byzantine process. Thus, of those READY messages, at most k are from Byzantine processes and, thus, at least $\lfloor (n - k)/2 \rfloor + 1 > 1$ are from correct processes. The result now follows from statement (4) of Sublemma A.1. □

We now prove Sublemmas A.2.2–A.6, which are used to prove the agreement property of consensus in Lemma A.5 below.

SUBLEMMA A.2. *If a correct process p decides on value e for round r, then at least $\lfloor (n - k)/2 \rfloor + 1$ correct processes have received from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes properly formed and justified* CONFIRM *messages for round r containing the value e and have updated their estimate to e, their timestamp to r and their confirms set to the set of those received* CONFIRM *statements.*

Proof. If a correct process p decides on value e for round r, it must have received properly formed and justified READY messages for round r containing the same value e from $\lfloor (n+k)/2 \rfloor + 1$ distinct processes, at least $\lfloor (n-k)/2 \rfloor + 1$ of which are correct processes. Each of those correct processes sends a properly formed and justified READY message containing the value e for round r only if it has received from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes properly formed and justified CONFIRM messages for round r containing the value e and has updated its estimate to e, its timestamp to r and its *confirms* set to the set of received CONFIRM messages. □

SUBLEMMA A.3. *If a correct process q decides on value e for round r, then the coordinator sends a properly formed and justified* SELECT *message for round r containing the value e.*

Proof. If a correct process q decides on value e for round r then, by Sublemma A.2, at least $\lfloor (n - k)/2 \rfloor + 1$ correct processes have received properly formed and justified CONFIRM messages for round r containing the value e from $\lfloor (n + k)/2 \rfloor + 1$ distinct processes and have updated their estimate to e, their timestamp to r and their *confirms* set to the set of received CONFIRM messages.

At least $\lfloor (n-k)/2 \rfloor + 1$ of the processes that sent properly formed and justified CONFIRM messages for round r are correct processes. Each of those correct processes sends a CONFIRM message containing the value e for round r only if it has received a properly formed and justified SELECT message for round r containing the value e. □

SUBLEMMA A.4. *A properly formed and justified* SELECT *message for round r with timestamp $ts > 0$ contains the unique value e contained in the* CONFIRM *messages for round ts from $\lfloor (n - k)/2 \rfloor + 1$ correct processes.*

Proof. A properly formed and justified SELECT message for round r with timestamp $ts > 0$ contains a properly formed ESTIMATE statement for round r with timestamp $ts > 0$ and value e. A properly formed and justified ESTIMATE message for round r with timestamp $ts > 0$ and value e contains a *confirms* set with properly formed CONFIRM statements for round ts containing the same value e from $\lfloor(n + k)/2\rfloor + 1$ distinct processes, of which $\lfloor(n - k)/2\rfloor + 1$ are correct processes. Any other set of $\lfloor(n + k)/2)\rfloor + 1$ processes would have at least $\lfloor(n - k)/2\rfloor + 1$ correct processes. This means that there would be at least one correct process that belongs to both sets, which is impossible. Thus, all properly formed and justified ESTIMATE messages with timestamp ts must contain the value e and a properly formed and justified SELECT message with timestamp ts must contain the value e. □

SUBLEMMA A.5. *If a correct process p decides on value e for round r and if the coordinator sends a properly formed and justified* SELECT *message for round r', $r' > r$, then that* SELECT *message has timestamp ts such that $r \le ts < r'$. Moreover, if $ts = r$, then the* SELECT *message contains the value e.*

Proof. If a correct process p decides on value e for round r then, by Sublemma A.2, at least $\lfloor(n - k)/2\rfloor + 1$ correct processes have received from $\lfloor(n + k)/2\rfloor + 1$ distinct processes properly formed and justified CONFIRM messages for round r containing the value e and have updated their estimates to e, their timestamps to r and their *confirms* sets to the CONFIRM messages that they received. Each such correct process sends an ESTIMATE message in task 1, phase 1 of round r', $r' > r$, with timestamp at least r, which contains its *confirms* set as justification.

In task 1, phase 2 of round r', the coordinator collects properly formed and justified ESTIMATE messages from $n - k$ distinct processes. At least one of those $n - k$ distinct processes is one of the $\lfloor(n - k)/2\rfloor + 1$ correct processes that sent an ESTIMATE message with timestamp at least r (because $\lfloor(n - k)/2\rfloor + 1 > k$). Consequently, the largest timestamp of the $n - k$ ESTIMATE messages for round r' is at least r. Moreover, the largest timestamp of the $n - k$ ESTIMATE messages is less than r', because it was set in a round prior to r'. Thus, the timestamp ts in a properly formed and justified SELECT message for round r' must satisfy $r \le ts < r'$.

Now suppose that the SELECT message for round r' has timestamp $ts = r$. By Sublemma A.4, a properly formed and justified SELECT message for round r' with timestamp ts contains the unique value e contained in the CONFIRM messages for round ts from at least $\lfloor(n - k)/2\rfloor + 1$ correct processes. This value e is process p's decision value as noted above. □

SUBLEMMA A.6. *If a correct process p decides on value e for round r and if the coordinator sends a properly formed and justified* SELECT *message for round r', where $r' > r$, then that* SELECT *message contains the value e and timestamp $ts > 0$.*

Proof. The proof is by induction on r'. In the base case, $r' = r + 1$. By Sublemma A.5, the timestamp ts in the SELECT message satisfies $r \le ts < r + 1$ and, thus, $ts = r > 0$. Moreover, the SELECT message contains the value e.

To establish the inductive step, we assume that the statement holds for all rounds r'', $r < r'' \le r'$, and show that it holds for round $r' + 1$. By Sublemma A.5, the properly formed and justified SELECT message for round $r' + 1$ contains timestamp ts, where $r \le ts < r' + 1$. Moreover, if $ts = r$, then the SELECT message contains the value e.

Consider now the case where $r < ts < r'+1$. The value e' in the SELECT message for round $r' + 1$ is the value e' in the properly formed and justified ESTIMATE message for round $r' + 1$ with timestamp ts, where ts is the largest timestamp in the $n - k$ ESTIMATE messages for round $r' + 1$ received by the coordinator. The value e' in a properly formed and justified ESTIMATE message with timestamp ts is the common value e' contained in properly formed and justified CONFIRM messages for round ts from $\lfloor(n+k)/2\rfloor+1$ distinct processes, at least $\lfloor(n - k)/2\rfloor + 1$ of which are correct processes. Each such correct process sends a CONFIRM message containing the value e' for round ts only if it has received a properly formed and justified SELECT message for round ts containing the value e'. By the inductive hypothesis, since $r < ts \le r'$, a properly formed and justified SELECT message for round ts contains the value e and, thus, $e' = e$. □

LEMMA A.5. (Agreement) *No two correct processes decide differently.*

Proof. If a correct process p decides on value e for round r and a correct process q decides on value e' for the same round r, then $e = e'$. To decide on e for round r, process p must have received properly formed and justified READY messages for round r containing the same value e from $\lfloor(n + k)/2\rfloor + 1$ distinct processes of which at least $\lfloor(n - k)/2\rfloor + 1$ are correct processes and, similarly, for q. Thus, both p and q must have received a properly formed and justified READY message for round r from the same correct process containing the same value $e = e'$.

If a correct process p decides on value e for round r and a correct process q decides on value e' for round r', where $r < r'$, then $e = e'$ by Sublemmas A.3, A.4 and A.6. □

Finally, we prove Lemma A.6, which establishes the irrevocability property of consensus.

LEMMA A.6. (Irrevocability) *Once a correct process decides on a value, it remains decided on that value.*

Proof. Once a correct process p decides on a value in task 3 of the algorithm, task 3 is terminated. Therefore, p does not make a subsequent decision. □

PROPOSITION A.2. *The consensus algorithm A_1 of Figure 3 solves the consensus problem using any fault detector in $\diamond\mathcal{S}(\text{Byz}, A_1)$ in an asynchronous distributed* *system with n processes in which the maximum number k of Byzantine faults satisfies $k \leq \lfloor (n-1)/3 \rfloor$.*

Proof. The proposition follows from Lemmas A.3, A.4, A.5 and A.6. □

THEOREM 7.1. *The consensus algorithm A_1 of Figure 3 solves the consensus problem using any fault detector in $\diamond\mathcal{W}(\text{Byz}, A_1)$ in an asynchronous distributed system with n processes, where the maximum number k of Byzantine faults satisfies $k \leq \lfloor (n-1)/3 \rfloor$.*

Proof. The theorem follows from Propositions A.1 and A.2. □

APPENDIX B

To establish that the algorithm given in Figure 5 is correct, we prove Theorem 8.1 below. This theorem holds in a partially synchronous system S that conforms to $\mathcal{M}_3$; that is, for every run of the algorithm A in S there is a GST after which some bounds on processing and message transmission times hold, although the values of GST and these bounds are not known *a priori*.

LEMMA B.1. *The algorithm given in Figure 5 satisfies eventual strong Byzantine completeness for algorithm A.*

Proof. We show that there is a time after which every process that has detectably deviated from algorithm A is permanently suspected by every correct process. Thus, suppose a process q detectably deviates from A. There are three possible cases as follows.

Case 1. Process q omits to send a required message m such that no correct process receives m, either directly or indirectly through message diffusion. Thus, there is a time t' after which all correct processes time out on message m. According to the algorithm, after time t' all correct processes permanently suspect q.

Case 2. Process q sends to a correct process p a message m that is not properly formed or is not properly justified according to the requirements of the algorithm. According to the fault detector algorithm, p sends m to all processes and, thus, there is a time after which all correct processes receive m and permanently suspect q.

Case 3. Process q sends to some correct process p a message m, and q also sends to some correct process s a message m', where m and m' are mutant messages. According to the algorithm, p sends m to all processes, s sends m' to all processes and, thus, there is a time after which all correct processes receive both m and m' and permanently suspect q. □

LEMMA B.2. *The algorithm given in Figure 5 satisfies eventual strong accuracy.*

Proof. We show that for any correct processes p and q, there is a time after which p does not suspect q. There are three possible cases as follows.

Case 1. Process p adds q to $byzset_p$ and, thus, suspects q permanently. According to the algorithm given in Figure 5, p must have received a message m sent by q that is not properly formed or is not properly justified or is a mutant of a message m' that q sent. However, this is impossible for a correct process. Thus, this case cannot occur.

Case 2. A finite number of timeouts, that were set by p for expected messages from q, expire. Suppose that p's last expired timeout on q is for message m. Because p and q are correct and q sends message m, eventually p receives m and permanently removes q from its suspects list.

Case 3. An infinite number of timeouts, that were set by p for expected messages from q, expire. Thus, infinitely many messages are added and removed from $expecting_p[q]$. According to the algorithm, every time a message m is removed from $expecting_p[q]$, the timeout period $\Delta_p(q)$ is increased. Thus, $\Delta_p(q)$ grows without bound. Eventually, the bounds on relative process speeds and message transmission times hold and $\Delta_p(q)$ is larger than the timeout based on these bounds. After this point, any timeouts that p sets on messages from q cannot expire, a contradiction to the assumption that an infinite number of timeouts expire. Thus, this case cannot occur. □

THEOREM 8.1. *The algorithm given in Figure 5 implements an eventually perfect Byzantine fault detector $\mathcal{D} \in \diamond\mathcal{P}(\text{Byz}, A)$ for a general algorithm A that uses a fault detector.*

Proof. The proof follows from Lemmas B.1 and B.2. □

Theorem 8.2 establishes that the algorithm given in Figure 6 is correct. This theorem holds in a partially synchronous system S that conforms to $\mathcal{M}_3$, that is, for every run of the consensus algorithm A_1 in S there is a GST after which some bounds on processing and message transmission times hold, although the values of GST and the bounds are not known *a priori*.

THEOREM 8.2. *The algorithm given in Figure 6 implements an eventually perfect Byzantine fault detector $\mathcal{D} \in \diamond\mathcal{P}(\text{Byz}, A_1)$, where A_1 is the algorithm given in Figure 3.*

Proof. The proof is based on lemmas similar to Lemmas B.1 and B.2. □

APPENDIX C

For our consensus algorithm and the latency degree of Schiper [18] defined in Section 9.1, we now prove the following theorem.

THEOREM 9.1. *The consensus algorithm A_1 given in Figure 3 has a latency degree of 4.*

Proof. In the optimal case, the first coordinator is correct and there are no suspicions. Each process decides in task 3 after it has received READY messages from $\lfloor (n+k)/2 \rfloor + 1$ processes. The statement now follows because:

(1) the ESTIMATE messages sent to the coordinator in task 1, phase 1 all carry a timestamp equal to 1;

(2) the SELECT message sent by the coordinator in task 1, phase 2 carries a timestamp equal to 2;
(3) the CONFIRM messages sent in task 2 all carry a timestamp equal to 3;
(4) the READY messages sent in task 1, phase 3 all carry a timestamp equal to 4. □

The expected number of broadcast messages required by the consensus algorithm is given in Theorem 9.2 below, which we prove using Lemmas C.1–C.4.

LEMMA C.1. *The expected number of broadcast messages sent in tasks one and two for one round of the consensus algorithm A_1 given in Figure 3 is $(3n + 1)$ in a system of n processes.*

Proof. The number of broadcast messages required by tasks one and two of the algorithm for one round with n processes is as follows.

Task 1, phase 1: n ESTIMATE messages.
Task 1, phase 2: 1 SELECT message.
Task 1, phase 3: n READY/NREADY messages.
Task 2, phase 1: n CONFIRM messages. □

In Lemmas C.2, C.3 and C.4 and Theorem 9.2, ρ denotes the probability that at least $\lfloor (n-k)/2 \rfloor$ processes falsely suspect a particular correct process that is acting as coordinator so that consensus is not reached and σ denotes the probability that at least $\lfloor (n-k)/2 \rfloor$ processes suspect a particular Byzantine process that is acting as coordinator so that consensus is not reached.

LEMMA C.2. *Let $\mathcal{P}(n, b)$ be the probability that consensus is reached in the current rotation with n processes of which b are Byzantine, given that consensus has not been reached in a prior rotation. Then $\mathcal{P}(n, b) = 1 - \rho^{n-b}\sigma^b$.*

Proof. For consensus to fail to be reached during the current rotation, every correct coordinator must be suspected by at least $\lfloor (n-k)/2 \rfloor$ correct processes and every Byzantine coordinator must be suspected by at least $\lfloor (n-k)/2 \rfloor$ correct processes. The probability that all of the $n - b$ correct coordinators are suspected by at least $\lfloor (n-k)/2 \rfloor$ correct processes is equal to ρ^{n-b}. The probability that all of the b Byzantine coordinators are suspected by at least $\lfloor (n-k)/2 \rfloor$ correct processes is equal to σ^b. Therefore, the probability that consensus is reached in the current rotation is $\mathcal{P}(n, b) = 1 - \rho^{n-b}\sigma^b$. □

LEMMA C.3. *Let $\mathcal{N}(n, b)$ be the expected number of rounds to reach a decision with n processes of which b are Byzantine, given that consensus is reached in the current rotation. Then*

$$\mathcal{N}(n, b) = \sum_{i=0}^{n-1} \sum_{j=0}^{i} \frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}} \rho^{i-j}\sigma^j.$$

Proof. First we consider the case in which $b = 0$, i.e. none of the processes is Byzantine. In this case, the expected number of rounds to reach consensus is

$$\begin{aligned}\mathcal{N}(n, 0) &= 1(1-\rho) + 2\rho(1-\rho) + 3\rho^2(1-\rho) \\ &\quad + \ldots + (n-1)\rho^{n-2}(1-\rho) + n\rho^{n-1} \\ &= 1 + \rho + \rho^2 + \rho^3 + \ldots + \rho^{n-1} \\ &= (1-\rho^n)/(1-\rho) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{i} \frac{\binom{n-i}{0-j}\binom{i}{j}}{\binom{n}{0}} \rho^{i-j}\sigma^j\end{aligned}$$

where the last term on the right-hand side of the first equality contains no factor of $(1-\rho)$ because of the precondition that consensus is reached during the current rotation.

Next, we consider the case in which $b = n$, i.e. all of the processes are Byzantine. In this case, the expected number of rounds to reach consensus is

$$\begin{aligned}\mathcal{N}(n, n) &= 1(1-\sigma) + 2\sigma(1-\sigma) + 3\sigma^2(1-\sigma) \\ &\quad + \ldots + (n-1)\sigma^{n-2}(1-\sigma) + n\sigma^{n-1} \\ &= 1 + \sigma + \sigma^2 + \sigma^3 + \ldots + \sigma^{n-1} \\ &= (1-\sigma^n)/(1-\sigma) \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{i} \frac{\binom{n-i}{n-j}\binom{i}{j}}{\binom{n}{n}} \rho^{i-j}\sigma^j\end{aligned}$$

where the last term on the right-hand side of the first equality contains no factor of $(1-\sigma)$ because of the precondition that consensus is reached during the current rotation.

In general for $n > 0$ and $0 < b < n$, the expectation $\mathcal{N}(n, b)$ for the number of rounds is the sum of the following four terms.

(1) The probability that the coordinator for this round is not Byzantine and is not falsely suspected by $\lfloor (n-k)/2 \rfloor$ processes and, thus, that consensus is reached during this round, multiplied by one round.
(2) The probability that the coordinator for this round is not Byzantine but is falsely suspected by $\lfloor (n-k)/2 \rfloor$ processes and, thus, that consensus is not reached during this round, multiplied by one round plus the expected number of rounds with one fewer $(n-1)$ processes and the same number (b) of Byzantine processes.
(3) The probability that the coordinator for this round is Byzantine and is suspected to be Byzantine by $\lfloor (n-k)/2 \rfloor$ processes and, thus, that consensus is not reached during this round, multiplied by one round plus the expected number of rounds with one fewer $(n-1)$ processes and one fewer $(b-1)$ Byzantine processes.
(4) The probability that the coordinator for this round is Byzantine and behaves in such a manner that it is not suspected by $\lfloor (n-k)/2 \rfloor$ processes and, thus, that consensus is reached during this round, multiplied by one round.

Thus

$$\begin{aligned}\mathcal{N}(n,b) &= \frac{n-b}{n}(1-\rho)1 + \frac{n-b}{n}\rho(1+\mathcal{N}(n-1,b)) \\ &\quad + \frac{b}{n}\sigma(1+\mathcal{N}(n-1,b-1)) + \frac{b}{n}(1-\sigma)1 \\ &= 1 + \frac{n-b}{n}\rho\mathcal{N}(n-1,b) + \frac{b}{n}\sigma\mathcal{N}(n-1,b-1).\end{aligned}$$

We now verify the general result. Starting with the right-hand side of the recurrence relation, we have

$$\begin{aligned}&1 + \frac{n-b}{n}\rho\mathcal{N}(n-1,b) + \frac{b}{n}\sigma\mathcal{N}(n-1,b-1) \\ &= 1 + \frac{n-b}{n}\rho\sum_{i=0}^{n-2}\sum_{j=0}^{i}\frac{\binom{n-i-1}{b-j}\binom{i}{j}}{\binom{n-1}{b}}\rho^{i-j}\sigma^{j} \\ &\quad + \frac{b}{n}\sigma\sum_{i=0}^{n-2}\sum_{j=0}^{i}\frac{\binom{n-i-1}{b-j-1}\binom{i}{j}}{\binom{n-1}{b-1}}\rho^{i-j}\sigma^{j} \\ &= 1 + \frac{n-b}{n}\rho\sum_{i=1}^{n-1}\sum_{j=0}^{i-1}\frac{\binom{n-i}{b-j}\binom{i-1}{j}}{\binom{n-1}{b}}\rho^{i-j-1}\sigma^{j} \\ &\quad + \frac{b}{n}\sigma\sum_{i=1}^{n-1}\sum_{j=1}^{i}\frac{\binom{n-i}{b-j}\binom{i-1}{j-1}}{\binom{n-1}{b-1}}\rho^{i-j}\sigma^{j-1} \\ &= 1 + \sum_{i=1}^{n-1}\sum_{j=0}^{i-1}\frac{\binom{n-i}{b-j}\binom{i-1}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^{j} \\ &\quad + \sum_{i=1}^{n-1}\sum_{j=1}^{i}\frac{\binom{n-i}{b-j}\binom{i-1}{j-1}}{\binom{n}{b}}\rho^{i-j}\sigma^{j} \\ &= 1 + \sum_{i=1}^{n-1}\sum_{j=0}^{i}\frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^{j} \\ &= \sum_{i=0}^{n-1}\sum_{j=0}^{i}\frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^{j} \\ &= \mathcal{N}(n,b).\end{aligned}$$

□

LEMMA C.4. *Let $\mathcal{R}(n,b)$ be the expected number of rounds for the consensus algorithm given in Figure 3 to decide on a value in a system with n processes of which b are Byzantine. Then*

$$\mathcal{R}(n,b) = n\frac{\rho^{n-b}\sigma^{b}}{1-\rho^{n-b}\sigma^{b}} + \sum_{i=0}^{n-1}\sum_{j=0}^{i}\frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^{j}.$$

Proof. The expected number of rounds to reach consensus is the sum of the following two terms.

(1) The probability that consensus is reached in the current rotation multiplied by the expected number of rounds to reach consensus, given that consensus is reached in the current rotation.
(2) The probability that consensus is not reached in the current rotation multiplied by the number of rounds n in this rotation plus the expected number of rounds to reach consensus following the end of this rotation.

Thus,

$$\mathcal{R}(n,b) = \mathcal{P}(n,b)\mathcal{N}(n,b) + (1-\mathcal{P}(n,b))(n+\mathcal{R}(n,b)).$$

Solving for $\mathcal{R}(n,b)$, we obtain

$$\mathcal{R}(n,b) = n\frac{1-\mathcal{P}(n,b)}{\mathcal{P}(n,b)} + \mathcal{N}(n,b).$$

Consequently, by Lemmas C.2 and C.3, the expected number of rounds to reach consensus is

$$\mathcal{R}(n,b) = n\frac{\rho^{n-b}\sigma^{b}}{1-\rho^{n-b}\sigma^{b}} + \sum_{i=0}^{n-1}\sum_{j=0}^{i}\frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^{j}.$$

□

Finally, we prove the following theorem, which gives the expected number of messages required to reach consensus.

THEOREM 9.2. *The expected number of broadcast messages needed by the consensus algorithm A_1 given in Figure 3 to decide on a value in a system with n processes of which b are Byzantine is*

$$(3n+1)\left(n\frac{\rho^{n-b}\sigma^{b}}{1-\rho^{n-b}\sigma^{b}} + \sum_{i=0}^{n-1}\sum_{j=0}^{i}\frac{\binom{n-i}{b-j}\binom{i}{j}}{\binom{n}{b}}\rho^{i-j}\sigma^{j}\right).$$

Proof. The result follows from Lemmas C.1 and C.4. □