Consensus in the Presence of Timing Uncertainty: Omission and Byzantine Failures

(Extended abstract)

Stephen Ponzio
MIT Laboratory for Computer Science

Abstract

We consider the time complexity of reaching agreement in a semi-synchronous model of distributed systems, in the presence of omission and Byzantine failures. In our semi-synchronous model, processes have inexact knowledge about the time to perform certain primitive actions: messages arrive within time d of when they are sent and the time between two consecutive steps of any process is in the known interval $[c_1, c_2]$. We use $C = c_2/c_1$ as a measure of the timing uncertainty.

A simple adaptation of the synchronous lower bound shows that at least time $(f+1)d$ is required to tolerate f failures; time $(f+1)Cd$ is sufficient for stopping or omission failures by directly simulating synchronous rounds. By strengthening the algorithm for stopping failures of Attiya, Dwork, Lynch, and Stockmeyer ([1]), we derive an algorithm for omission failures that has minimal dependency on the uncertainty factor C. If fewer than half the processes are faulty then the running time is $4(f+1)d + Cd$, which is within a factor of 4 of optimal and may be much faster than direct rounds simulation if C is large. If more than half the processes are faulty, then the running time is shown to be greater by approximately a factor of $\min(\frac{f}{n-f}, \sqrt{C})$.

Finally, we present a general simulation for $n \geq 3f+1$ that tolerates Byzantine failures and simulates any round-based synchronous algorithm at a cost of time $2Cd + d$ per round.

1 Introduction

Real systems are likely to be neither perfectly synchronous nor completely asynchronous. Recent work in analyzing realtime systems has made an effort to more accurately measure the amount of real time used by real systems when the time needed to perform certain primitive steps is known only approximately.

We are interested in studying how the degree of *timing uncertainty* affects the time complexity of distributed computing problems. Of particular interest are problems that are intractable in an asynchronous setting yet have solutions with tight bounds in the synchronous setting; a natural candidate is the consensus problem. In our semi-synchronous model, processes have uncertain information about the time needed to perform certain primitive operations: every message is delivered within time d of when it is sent and the amount of time between any two consecutive steps of any process is in the known interval $[c_1, c_2]$.[1] We use $C = c_2/c_1$ as a measure of the timing uncertainty. We assume that processes begin executing a given algorithm at the same time and measure the amount of real time until all nonfaulty processes reach a decision. In this paper, we consider two kinds of process failure: send-omission failures and

[1]Results of [14] and [8] imply that if any of the bounds d, c_1, and c_2 does not exist then there is no algorithm for consensus tolerating even a single stopping failure.

Byzantine (arbitrary) failures.

The time complexity of the consensus problem has been well studied in the synchronous "rounds" model (see, for example, [16, 18, 13, 10, 7]). The synchronous lower bound of $f+1$ rounds ([17, 12, 5]) can be adapted in a straightforward way to yield a lower bound of $(f+1)d$ in our semi-synchronous model. For omission failures, an upper bound of approximately $(f+1)(Cd+d)$ is achievable by directly simulating the rounds of any synchronous algorithm, at a cost of time $Cd+d$ per round. This factor of C arises from detecting the absence of messages. If another process fails to send a message at time t, the recipient cannot be sure that the message will not arrive until time d has elapsed. Because process steps are the only actions for which a lower bound on time is known, a process must wait for d/c_1 of its own steps in order to be certain that time d has elapsed (in case that it is running "fast" with time c_1 between steps), which may take time $c_2(d/c_1) = Cd$ (if it is running "slowly"). Thus each round takes approximately time Cd to simulate. When failures are arbitrary, it is not clear even how to simulate a synchronous algorithm correctly.

In [1], Attiya, Dwork, Lynch, and Stockmeyer prove nearly tight upper and lower bounds on the time to reach consensus in the presence of stopping failures. By combining several lower bound techniques, they prove a lower bound of $(f-1)d+Cd$. More surprisingly, they give a clever algorithm for consensus that runs in time $2fd+Cd$, much faster than a direct simulation when C is large.

This paper makes three contributions: the strengthening the algorithm of [1] to tolerate omission failures requiring a finer analysis, the characterization both algorithms as the simulation of a simple synchronous algorithm, and a general simulation tolerant of Byzantine failures. The resulting algorithm for omission failures has running time $4(f+1)d+Cd$ for $n \geq 2f+1$, which is within a constant factor (4) of the lower bound, minimizing the dependence on the timing uncertainty C. If $n \leq 2f$, then a more involved analysis proves two different upper bounds for the running time: $(3\frac{f}{n-f}+5)(f+1)d+Cd$ and $(2\sqrt{C}+6)(f+1)d+Cd$. We identify our algorithm and that of [1] as optimized simulations of a synchronous algorithm. This view may help in understanding what problems can and cannot be efficiently solved in this semi-synchronous model. Our general simulation tolerates Byzantine failures and works for any round-based synchronous algorithm for $n \geq 3f+1$ at a cost of time $2Cd+d$ per round, implying a consensus algorithm for our semi-synchronous model requiring time $(f+1)(2Cd+d)$.

1.1 Related work

In [11], the consensus problem was studied using a model of partial synchrony in which upper bounds on message delivery time and/or processes' relative step rates exist, but are unknown a priori to the processes. The algorithms of [11] are concerned with fault tolerance rather than timing efficiency, and therefore translate to relatively slow algorithms for our model. In contrast, we concentrate on exact time complexity. Current work also concentrating on the real time complexity of the consensus problem appears in [21]. There, process clocks are assumed to be synchronized to within a fixed additive error. In contrast to our results, the results there are stated in terms of process clock time, not absolute time. It is unclear how to translate results back and forth between their model and ours; this is posed as a direction for further research.

A related model is proposed in [15] to study the time complexity of detecting failures along a network path. This model assumes synchronous processes but differentiates between the worst-case a priori bound Δ on message delay and the actual worst-case message delay δ in a given execution, which may be much less. It is thus desirable for algorithms to have minimal dependency on Δ. This model raises similar concerns as our model does: detecting the absence of a message may be much more expensive than receiving the message. It is not difficult to see that direct implementation of synchronous consensus algorithms gives a running time of about $(f+1)\Delta$ for any type of failures; on the other hand, clearly the synchronous lower bound implies that no algorithm can guarantee a running time of less than $(f+1)\delta$. In this model, our algorithms yield an improvement over direct simulation strategies similar to the corresponding improvement in our semi-synchronous model. In fact, our algorithms may be run without modification (with $c_1 = c_2$) in the model of [15], yielding the running times derived here with the syntactic substitution of δ for d and Δ/δ for C.

Other work in this area includes the extensive literature on clock synchronization algorithms (see [20] for a survey). Algorithms from [22] and the previously mentioned consensus work of [11] include ideas similar to those in our simulation tolerant of arbitrary failures. Other problems recently studied in our model of timing uncertainty include the problem of mutual exclusion [2] and the complexity of a network synchronizer algorithm [3].

2 Model

Our underlying formal model is essentially the same as that used in [1]. Our model differs by assuming for ease of presentation that all processes begin executing the algorithm at time 0 and all messages are delivered in the order sent. The latter assumption is easily removed from both of our algorithms; for our omissions algorithm, the former may be replaced by the formalism of [1] whereby processes receive "input(v_i)" actions and running time is measured beginning with the latest of these.

We consider a system of n processes $1, \ldots, n$. Each process is modeled as a deterministic (possibly infinite) state machine. Processes communicate by sending messages via a completely connected and totally reliable message system. Formally, we model a computation of the algorithm as a sequence of *configurations* alternated with *events*. A configuration is simply a vector of the processes local states. Events are of two types, process steps and message delivery events, and each event has a real time associated with it. During a process step, a process performs local computation and may send messages. During a message delivery event, a process does not take a step but may change its state according to the message delivered (e.g., "remember" the delivered message). We consider executions of the system in which the successive steps of all nonfaulty process are separated by at least time c_1 and at most c_2, and all messages sent are delivered within time d. A *timed execution* α is an execution of the algorithm satisfying these timing requirements. Although it is not necessary for the proof, we will generally assume that $c_2 \ll d$: process step time is very small compared to the maximum message delay time. In the analysis, we therefore approximate the quantity $d + c_2$ by d.

Omission failures. A process suffers an *omission failure* if at some step of the process, the messages it sends are a subset of the messages determined by the algorithm. If the algorithm requires j to broadcast a message to all processes, but j does not send a message to i, then we say that "j omits to i" or that this broadcast is "unsuccessful".

Byzantine failures. A process that suffers a *Byzantine failure* may exhibit arbitrary behavior; no restrictions are made on its state transitions or what messages it sends. Also, the time between successive steps of a faulty process might *not* be in the interval $[c_1, c_2]$.

Consensus. Finally, we briefly define the consensus problem. Each process receives as input a binary value as part of its local state. The problem is for the processes to agree on a binary value despite the failure of some processes. We say that a timed execution α is *f-admissible* if at most f processes fail in α. An algorithm *solves the consensus problem for f failures within time T* provided that for each of its *f-admissible* timed executions α, (1) no two different processes decide on different values (agreement), (2) if some nonfaulty process decides on v, then some process has initial value v (validity), and (3) every nonfaulty process decides by time T (time bound).

We consider the binary version of the problem, where the initial values are 0 or 1. Like the algorithm of [1], our algorithm for omission failures can be extended to work for any value set, using the same extension given there ([1], §5.4). Our algorithm for Byzantine failures is a general simulation for any rounds based algorithm and therefore can simulate any synchronous agreement algorithm for any value set.

3 Omissions failures

In this section, we strengthen the algorithm of [1] to tolerate omission failures. Unlike [1], our algorithm requires an a priori bound f on the number of failures to be tolerated. It is, however, "early stopping": the running time for a given execution is a function of the number of processes that fail in that execution, not of the maximum number of failures to be tolerated.

3.1 Intuition: the underlying synchronous algorithm

Our algorithm and the algorithm of [1] may be interpreted as simulations of the following simple synchronous algorithm:

```
ROUND 0:
    If $v = 1$, then send "I didn't decide in ROUND 0"
                and goto ROUND 1.
    If $v = 0$, then send "I decided in ROUND 0"
                and decide 0.
ROUND $r > 0$:
    If "I decided in ROUND $r - 1$" received,
      then send "I didn't decide in ROUND $r$"
           and goto ROUND $r + 1$
    If no "I decided in ROUND $r - 1$" received,
      then send "I decided in ROUND $r$"
           and decide $r$ mod 2.
```

It is easy to see that if a nonfaulty process decides in round r then no process decides in round $r + 1$ and all processes then decide in round $r + 2$. The algorithm is also "early-stopping": any execution in which at most f processes fail takes at most $f + 2$ rounds of communication. (This means that all processes decide in round $f+2$ or earlier, despite the fact that the first round is numbered 0, since a decision in round i is based on messages sent in round $i - 1$ or earlier.) The is easily seen by observing that if an execution takes x rounds then a faulty process decides in each of rounds 0 through $x-3$: if no faulty process decides in round $i \leq x-3$ then either (1) a nonfaulty process decides in round i and all processes decide by round $i + 2$, or (2) no process decides in round i and therefore they all decide in round $i + 1$ (because no process receives an "I decided in round i" message).

Both our algorithm and that of [1] simulate this synchronous algorithm, making several important optimizations in order to improve the running time for our model. If during the simulation of round r, a process receives a message saying "I decided in round $r - 1$", it does not wait for round $r - 1$ messages from other processes, but immediately advances to round $r + 1$, broadcasting to all processes, in effect, "I *know of a process that* decided in round $r-1$". Other processes in round r that receive this message relay it to all processes and also advance immediately to round $r + 1$. A process may decide in round r only if it can be sure that no nonfaulty process decided in round $r - 1$. This is ascertained only when, for every other process p, either (1) the message "I didn't decide in round $r - 1$" is received from p, or (2) p has been detected as faulty, or (3) p has decided in some round $r' < r - 1$.

The key to the improved efficiency of our algorithm relative to that of [1] is the addition of a mechanism for a process to detect its own failure. We require that a process receive at least $n - f$ acknowledgments for every message of the synchronous algorithm that it sends. Until a process has received a sufficient number of acknowledgments for its round r message, it is prohibited from deciding in round $r+1$ or advancing to round $r+2$. This is important to the efficiency of the algorithm because it limits to 1 the number of times a faulty process can omit a message of the synchronous algorithm to all nonfaulty processes.

3.2 The algorithm

We first explain the presentation of our algorithm. We describe our algorithm as the parallel composition of a fault-detection protocol and a main algorithm. At each step, a process first executes the code of the fault-detection protocol, then executes the code of the main algorithm, and finally sends a message. (Recall that in our model a process may send at most one message at each step).

This message is the concatenation of possibly several component "messages" which are specified by the **queue** commands in the code: if during a step, the statement "**queue** 'm'" is executed in the code, then "message" m is a component of the message sent at the end of that step. We will refer to a message by any one of its components: we will say "an m message" or simply "an m" to refer to any message with m as one of its components. An omission failure causes the entire message to be omitted; the message cannot be corrupted by losing only some components.

Our model also specifies that a process receives messages only during delivery events (and therefore only between process steps). For every delivery event, a process changes its state by adding the received message to a buffer (an unordered set). At its next step, the process reads and empties this

```
STEP $s$:   If "shutdown $i$" received, then halt.
            If decided, then queue "I've decided: $s$"
                      else queue "I'm alive: $s$".
            For each $j \notin D \cup F$,
                if "shutdown $j$" message received
                  then $F \leftarrow F \cup \{j\}$; queue "shutdown $j$"
                if "I'm decided: $s_j$" message received from $j$
                  then $D \leftarrow D \cup \{j\}$
                if "I'm alive" messages from $j$ not numbered consecutively
                  then $F \leftarrow F \cup \{j\}$; queue "shutdown $j$"
                if no message received from $j$ and more than
                    $(d + c_2)/c_1$ steps taken since last message received from $j$,
                  then $F \leftarrow F \cup \{j\}$; queue "shutdown $j$".
```

Figure 1: The fault-detection protocol for process i at step number s.

buffer. A conditional statement in the code referring to the receipt of a message checks whether such a message was read from this buffer during the given step.

For ease of presentation, some components of a process's state are not explicitly named or maintained in the code—for instance, the number of steps a process has taken, whether it has decided, or whether it has sent a certain message. Process index subscripts are omitted in the code but used in the text (e.g., "D_i") to refer to a local variable (D) of process i.

3.2.1 The fault-detection protocol

A process sends a message at every step that it takes, consecutively numbering all messages that it sends with the number s of its current step.[2] Before a process decides, it sends the message "I'm alive: s" where s is the number of its current step; after a process decides, it sends "I've decided: s". The failure of a process can thus be detected by a gap in the sequence numbering (recall we assume that message links deliver messages in the order sent) or by the absence of any messages for too long a period of time (more than time $d + c_2$).

All processes detected as faulty are added to a local set F. When a process i detects the failure of another process j, it broadcasts this fact in the form of a "shutdown j" message. Upon receiving this message, other processes add j to their respective sets F; when process j receives this message, it *halts*, ceasing its execution of the algorithm. The timeout protocol also keeps track of which processes have decided. When a process receives a message "I've decided: s" from another process, it adds that process to its set D. When a process i adds j to D_i (F_i, resp.), we say it has "detected" that j has decided (failed, resp.). We say that a process i is *shut down at time t* if it receives a "shutdown i" message at time t. The code for the fault-detection protocol is in Figure 1.

We now state without proof two basic properties of the fault-detection protocol with respect to arbitrary executions.

Lemma 3.1 *If process i does not fail, then i is not added to any set F_j and is not shut down.* ∎

Lemma 3.2 *If at time t, process j omits a message to i, and i is not shut down by time $t + C(d + c_2) + (d + c_2) \approx Cd + d$, then i adds j to F_i by that time.* ∎

3.2.2 The main algorithm

The main algorithm is basically an asynchronous version of the synchronous algorithm of Section 3.1. Because in the synchronous algorithm, both "I decided in round r" and "I didn't decide in round

[2] As a consequence of the bound on running time to be derived, these sequence numbers are bounded by a function of f, d, c_1 and c_2.

```
PHASE 0:      If $v = 1$, then queue "0" and goto PHASE 1.
              If $v = 0$, then queue "1" and decide 0 and goto PHASE 2.

PHASE $r > 0$: For each $j$ and each $r'$,  $1 \leq j \leq n$ and $0 \leq r' < r$,
                 if "$r'$" message received from $j$,
                   then $M^{r'} \leftarrow M^{r'} \cup \{j\}$
                 if "ack($i, r-1$)" received from $j$ and $j \notin F$,
                   then $A^{r-1} \leftarrow A^{r-1} \cup \{j\}$
                 if $j \in M^{r'}$ and $r' < r$ and "ack($j, r'$)" not yet sent,
                   then queue "ack($j, r'$)"                      (whether decided or not)
              If decided and $M^{r-2} \neq \emptyset$ and "$r-2$" not yet sent,
                then queue "$r-2$"
              If not decided and $|A^{r-1}| \geq n - f$,             (enough ack's received)
                then if  $M^r \neq \emptyset$,                (some process decided in phase $r-1$)
                       then queue "$r$" and goto PHASE $r+1$
                     if $M^r = \emptyset$ and $j \in M^{r-1}$ for all $j \notin (D \cup F)$,
                       then queue "$r+1$" and decide $r$ mod 2 and goto PHASE $r+2$
```

Figure 2: The main algorithm, performed by a process at every step.

$r+1$" serve the same purpose—both mean "I know of a process that decided in round r"—our main algorithm uses the message "$r+1$" in place of both.

The code for the main algorithm appears in Figure 2. We call the simulation of round r of the synchronous algorithm "phase r". Each process i starts in phase 0 with v_i set to its own private value (1 or 0). As with the synchronous algorithm, in even numbered phases a process can decide only 0, and in odd numbered phases a process can decide only 1.

When a process advances from phase r to phase $r+1$, it broadcasts an "r" message. (This is the equivalent of the message "I didn't decide in round r" in the synchronous algorithm.) When a process decides in phase r, it broadcasts an "$r+1$" message. Set M^r contains those processes from which an r message has been received. A process may decide in phase r only if it has (1) not yet received an r message, and therefore does not know of a process that decided in round r, and (2) has received an $r-1$ message from all processes not yet detected as failed or decided, indicating that they did not decide in round $r-1$. If process i is nonfaulty, then the receipt of an $r+1$ message from i prevents other processes from deciding in phase $r+1$ since they do not add i to D or F before receiving it. A process that decides in round r does not send an r message unless it receives one first (this necessarily implies that some process decided in round $r-1$ but failed).

Our convention of acknowledging messages works as follows. Each process maintains a set A^r containing those processes from which a properly sequenced "ack($\cdot, r$)" message (i.e., sender not in F) has been received. (This restriction is necessary only for the bound when $n \leq 2f$.) Until a process decides, it sends exactly one acknowledgment message, "ack(j, r')", for each r' message it receives where r' is less than its current phase number. After a process decides in some phase r, it continues to acknowledge r' messages for $r' \leq r+1$. This is implemented in the code by allowing the process to advance to phase $r+2$ but no further. It is not necessary for a process to acknowledge r' messages for $r' > r+1$ because as we will see, if it is nonfaulty then other nonfaulty processes do not advance to phase $r+3$ without deciding and therefore do not require acknowledgments for their $r+2$ messages. Until a process has received at least $n-f$ properly sequenced acknowledgments for its r message ($|A^{r-1}| \geq n-f$), it may not advance to phase $r+1$ or decide in phase r.

Definition 1 *A process i is* blocked in phase r *(for $r > 0$) if it advances to phase r without deciding and never has $|A_i^{r-1}| \geq n-f$.*

Being blocked is a permanent state, but even if a process is not blocked in phase r, it may be temporarily *delayed* from advancing to phase $r+1$ as it waits for acknowledgments before proceeding.

We prove here a few basic lemmas about the main algorithm with respect to any f-admissible execution. The first two lemmas affirm two expected properties that held for the synchronous algorithm.

Lemma 3.3 *If some nonfaulty process decides in phase $r \geq 0$ then no process decides in phase $r+1$.*

Proof: Let i be a nonfaulty process that decides in phase r and consider any other process j. According to the code of the main algorithm, j cannot decide in phase $r+1$ without receiving an r message from i or adding i to F_j or D_j. Because i is nonfaulty, by Lemma 3.1 it is never added to F_j. During any step at which j has received an r message or an "I've decided" message from i, j must also have received an $r+1$ message from i and therefore is precluded from deciding in phase $r+1$ (which requires $M^{r+1} \neq \emptyset$). ■

The following definition is useful in proving correctness and analyzing time complexity.

Definition 2 *Phase r is* quiet *if there is some process that never receives any r messages.*

Lemma 3.4 *If a nonfaulty process decides in phase $r \geq 0$ then phase $r+2$ is quiet.*

Proof: By Lemma 3.3, no process decides in phase $r+1$. If a process does not decide in phase $r+1$, then it does not send an $r+2$ message until it receives one. Therefore, no process sends an $r+2$ message and in fact *no process* receives an r message. ■

The next two lemmas affirm that the convention of acknowledging r messages works as expected—nonfaulty processes are never blocked—and the last lemma states that the failure of blocked processes is eventually detected by all processes.

Lemma 3.5 *If a process i advances to phase $r \geq 1$ without deciding and sends an r' message to a nonfaulty process j for $0 \leq r' \leq r-1$, then i receives an "ack$(i, r-1)$" message from j.*

Proof: By induction on r. Clearly the lemma is true for $r=1$: j advances to phase 1 during its first step and sends "ack$(i,0)$" during the next step at which it has received a 0 message from i.

Assume the lemma is true for $r-1 \geq 1$. First observe that j does not decide in any phase $r' \leq r-3$: by Lemma 3.3, this would imply that no process decides in phase $r'+1$ and therefore no process sends an $r'+2$ message, but this is not possible because i advances to phase $r \geq r'+3$ without deciding and therefore must receive an $r'+2$ message. If j decides in phase r' and $r' = r-2$ or $r-1$, then j immediately advances to phase $r'+2 \geq r$ after deciding and sends "ack$(i, r-1)$" to i. Suppose that j does not decide in any phase $r' \leq k-1$. Process j must advance from each phase $r' \leq k-1$ because it is never shut down (by Lemma 3.1), has $M_j^{r'} \neq 0$ (an r' message is received from i), and has $|A_j^{r'}| \geq n-f$ (because it is nonfaulty and therefore sends an r'' message to all processes for each $r'' \leq r'-1$ and by the induction hypothesis receives "ack(j, r'')" from all nonfaulty processes—none of which, by Lemma 3.1 are ever added to F_j). Process j therefore advances to phase r and may then send "ack$(i, r-1)$" to i. ■

Corollary 3.6 *If process i is nonfaulty and advances to phase $r \geq 1$ without deciding, then it eventually has $|A_i^{r-1}| \geq n-f$. (A nonfaulty process is never blocked.)*

Proof: Because i is nonfaulty and advances to phase r without deciding, for $0 \leq r' < r$ it sends an r' message to all processes as it advances to phase $r'+1$. By Lemma 3.5, i receives "ack$(i, r-1)$" from each nonfaulty process. Because by Lemma 3.1, nonfaulty processes are never added to F_i, each nonfaulty process is added to A_i^{r-1}, giving the necessary bound. ■

The following lemma relies on the fact that a process continues to take steps, executing the algorithm after it decides; in particular, it continues to detect the failure of processes and, if necessary, send acknowledgments.

Lemma 3.7 *If a faulty process j unsuccessfully broadcasts an r message at time t and is subsequently blocked in phase $r+1$, then all processes* *not shut down by time $t + C(d + c_2) + 2(d + c_2) \approx t + Cd + 2d$ detect the failure of j by that time.*

Proof: By the definition of being blocked, j advances to phase $r + 1$ but never has $|A_j^r| \geq n - f$. Thus there is some nonfaulty process i never added A_j^r. By Lemma 3.5, j omits an r' message to i for some $0 \leq r' \leq r$. This omission occurs at or before time t. By Lemma 3.2, i detects this failure by time $t + C(d + c_2) + (d + c_2)$, broadcasting "shutdown j" to all processes in the same step. By time $d + c_2$ later, all processes not yet shut down have received this message and taken a step, adding j to their failed sets. ■

3.3 Proof of correctness

We now prove that in all f-admissible executions, the algorithm terminates and correctly satisfies the agreement and validity conditions. We first prove "progress"—that processes in fact advance to successive phases as expected. Given this progress lemma and a few simple facts about quiet phases, the proofs of agreement, validity, and termination are easily derivable. These proofs follow the same reasoning as the informal argument about the synchronous algorithm given in Section 3.1.

Lemma 3.8 *For each $r \geq 0$ and each process i that is neither blocked nor shut down in any phase $r' \leq r$, process i either decides in some phase $r' \leq r$ or advances to phase $r + 1$.*

Proof sketch: For contradiction, let phase r be the first phase for which the lemma is not satisfied and let i be any process for which the lemma is not satisfied at phase r. By the choice of r, i advances to phase r. First note that $r \neq 0$, since every process either decides or advances to phase 1 during its first step. Next note that i receives no r messages, since that would enable it to advance to phase $r+1$ (it is not blocked in phase r). It is not difficult to show that by the choice of r, if i receives no r messages, then for every other process j, i eventually either receives an $r - 1$ message from j or adds j to F_i or D_i. This enables p_i to decide in phase r, a contradiction. ■

Corollary 3.9 *For any $r \geq 0$, every nonfaulty process either decides in phase $r' \leq r$ or advances to phase $r + 1$.*

Proof: By Lemmas 3.1 and 3.6, a nonfaulty process is never shut down or blocked; the corollary then follows immediately from Lemma 3.8. ■

Corollary 3.10 *If phase $r \geq 0$ is quiet, then each nonfaulty process decides in some phase $r' \leq r$.*

Proof: By Corollary 3.9, each nonfaulty process either decides in phase $r' \leq r$ or advances to phase $r + 1$. But a nonfaulty process cannot advance to phase $r + 1$: to do so, it would send an r message to all processes, contradicting the assumption that phase r is quiet. ■

Lemma 3.11 (Agreement) *No two nonfaulty processes decide on different values.*

Proof: Let r be the first phase in which some nonfaulty process i decides. By Lemma 3.3, no process decides in phase $r + 1$. Because no process decides in phase $r+1$, no process sends an $r+2$ message and thus phase $r + 2$ is quiet. Thus by Lemma 3.10, all nonfaulty processes decide in some phase $r' \leq r+2$. By the choice of r, all nonfaulty processes decide in either phase r or phase $r + 2$, in either case on $r \bmod 2$. ■

Lemma 3.12 (Validity) *If any process decides on value b, then some process i starts with $v_i = b$.*

Proof: Clearly if some process j decides on 1, it does so in phase $r > 0$ and that process itself must have started with $v_j = 1$ since otherwise it would have decided on 0 during its first step.

If some process j decides on 0, it cannot be that all processes started with $v_i = 1$. For then, no process would decide in phase 0 and no process would send a 1 message. No process would receive a 1 message and therefore no process would advance to phase 2 without deciding and so no process would decide 0. ■

Lemma 3.13 (Termination) *In any f-admissible execution, there is a quiet phase numbered at most $f+2$ and so each nonfaulty process decides in some phase $r \leq f + 2$.*

Proof: If some nonfaulty process decides in phase $r \leq f$ then no process decides in phase $r + 1$ and no process sends an $r + 2$ message. Phase $r + 2$

is therefore quiet and by Lemma 3.10 all nonfaulty processes decide by phase $r+2 \leq f+2$.

If no nonfaulty process decides in any phase $r \leq f$, then there must be a phase $h-1$, $0 \leq h-1 \leq f$, in which no *faulty* process decides, and therefore in which *no* process decides. If a process does not decide in phase $h-1$, then it does not send an h message until it receives one. Therefore no process sends an h message—phase h is quiet—and by Lemma 3.10, all nonfaulty processes decide by phase $h \leq f+1$. ■

3.4 Analysis of time bounds

We now outline analysis of the algorithm's running time. The analysis that follows is carried out with respect to any f-admissible execution. We calculate an upper bound on the amount of real time until all nonfaulty processes decide in any such execution. Having already proved the correctness of the algorithm, we will hereafter assume $d \gg c_2$ and make approximations appropriately. We first introduce some notation.

- For $r \geq 0$, let t_r be the earliest time by which all processes not blocked in any phase $r' \leq r$ of the execution have either decided, advanced to phase $r+1$, or been shut down.
  Because every process either decides or advances to phase 1 on its first step, $t_0 = 0$.

- Let phase h be the first quiet phase.

- For $r \geq 0$, let $B_r = \{i : i \text{ is blocked in phase } r+1\}$; let $b_r = |B_r|$.

The definition of B_r may seem unusual, but it serves a purpose. We will want to bound $t_r - t_{r-1}$, which we think of as the time for phase r, in terms of the number of processes that omit an r message to all nonfaulty processes. This number is b_r, since all such processes are subsequently blocked in phase $r+1$. Note that for $r \neq r'$, $B_r \cap B_{r'} = \emptyset$ and therefore $\sum_r B_r \leq f$.

We prove an upper bound for two kinds of phases: those that are quiet and those that are not. A quiet phase may take up to time approximately Cd as processes may have to perform "long timeouts", but there can be only one quiet phase before all nonfaulty processes decide. We will show that the time of non-quiet phases does not depend on C.

A useful fact, which we here state without proof is that if a process receives a sufficient number of acknowledgments for its r message, then it receives them promptly, by time $t_{r-1}+2d$. (An r message is sent no later than time t_r and an acknowledgment for it is sent no later than $t_r + d$.)

Lemma 3.14 *If process j eventually has $|A_j^r| \geq n-f$, then it has $|A_j^r| \geq n-f$ by time $t_r + 2d$.* ■

To prove the bound for quiet phases, we use a proof similar to the proof of Lemma 3.8 to give a generous bound that holds for any phase:

Lemma 3.15 *$t_1 - t_0 \leq Cd + d$ and for any phase $r > 1$, $t_r \leq \max(t_{r-1} + Cd + d,\ t_{r-2} + Cd + 2d)$.*

Proof sketch: By time t_{r-1}, every process j either sends an $r-1$ message or an "I've decided" message or else omits a message to a nonfaulty process. If j omits a message to a nonfaulty process by then, its failure is detected by all processes by the greater of times $t_{r-1}+Cd+d$ and $t_{r-2}+Cd+2d$. If a process is not blocked in phase r, then it has $|A^r| \geq n-f$ by time $t_{r-1}+2d \leq t_{r-1}+Cd+d$. Thus, if a process is not blocked in phase r and has not decided or been shut down by the claimed time, it may advance to phase $r+1$ or decide (depending on whether or not it has received an r message). ■

In bounding the time of a phase r that is not quiet, we will bound the time until every process receives an r message (which every process does, by the definition of a quiet phase). By that time, every process that is not yet decided or shut down or blocked in any phase $r' \leq r$ may advance to phase $r+1$; thus this is a bound for t_r. In bounding the time until every process receives an r message, the following reasoning is at the heart of the analysis. In order for the first r message to ever be sent, some process must decide in phase $r-1$, which by definition, it does by time t_{r-1}. An r message sent by any other process i that does not decide in phase $r-1$ is sent because i received an r message. Thus, a causal chain of r messages may be followed back to the process that originated it, sending the "first" r message before t_{r-1}. Because after time $t_{r-1}+2d$ a process broadcasts an r message as soon as it receives one (at its next step), our time bound for phases that are not quiet is approximately d times the length of the shortest such chain to each process.

3.4.1 Bound for $n \geq 2f+1$

If $n \geq 2f+1$, we can be sure that when a faulty process broadcasts an r message, it either sends to at least one nonfaulty process or it sends to at most $f < n-f$ processes and therefore becomes blocked in phase $r+1$. If it sends to a nonfaulty process, then that process will send an r message to all processes. The number of processes blocked in phase $r+1$ is exactly b_r; our bound is roughly $b_r \cdot d$.

Lemma 3.16 *If phase $r \geq 1$ is not quiet and no nonfaulty process decides in any phase $r' \leq r$, then $t_r - t_{r-1} \leq (3+b_r)d$.* ∎

Instead of giving the proof of this lemma, we reinforce the above informal argument by describing how this bound is realized by a worst-case execution. Process $1 \in B_r$ is the first to send an r message. It decides in phase $r-1$ at time t_{r-1} (no later, by definition of t_{r-1}, since process 1 is not blocked in any phase $r' \leq r-1$) and sends an r message to only process $2 \in B_r$. Process 2 waits until time $t_{r-1}+2d$ for $|A_2^{r-1}| \geq n-f$ and then, having received an r message from 1, advances to phase $r+1$, sending an r message to only process $3 \in B_r$. The pattern is repeated until process $b_r+1 \notin B_r$ receives an r message at time $t_{r-1}+(b_r+1)d$. Process b_r+1 advances to phase $r+1$ and sends an r message to all processes except i. All nonfaulty process except i receive an r message from b_r+1 at time $t_{r-1}+(b_r+2)d$ and i receives an r message from them at time $t_{r-1}+(b_r+3)d$. By this time, each process has either advanced to phase $r+1$ (as it sent an r message), decided, been shut down, or is blocked in some phase $r' \leq r$.

We can now tightly bound the running time of any f-admissible execution by summing the bounds for all phases in that execution. The bound shows that the algorithm depends on C only to the extent of an additive factor of Cd. For C large, this algorithm may be far more efficient that a direct rounds simulation. The bound we obtain for $n \geq 2f+1$ is within approximately a factor of 4 of optimal: our bound is $4(f+1)d+Cd$; the lower bound proved in [1] is $(f-1)d+Cd$.

Theorem 3.17 *For $n \geq 2f+1$, the algorithm above solves the consensus problem for f omission failures within time $4(f+1)d+Cd$.*

Proof: For any given execution, let h be the first quiet phase. By Lemma 3.10, each nonfaulty process decides in some phase $r \leq h$, by time t_h. Clearly, if $h=0$ or $h=1$, we have the desired bound. If $h>1$, then we can bound the time for phases $1,\ldots,h-1$ by Lemma 3.16, and the time for phase h by Lemma 3.15. Thus we have

$$\begin{aligned}
t_h - t_0 &= \sum_{r=1}^{h-1}(t_r - t_{r-1}) + (t_h - t_{h-1}) \\
&\leq \sum_{r=1}^{h-1}(3+b_r)d + (Cd+d) \\
&\leq (f+1)3d + f\cdot d + (Cd+d) \\
&\quad \text{since } h \leq f+2 \text{ and } \textstyle\sum_r b_r \leq f \\
&= 4(f+1)d + Cd.
\end{aligned}$$

∎

We note that for $C \geq 4$, it is possible to construct an execution that takes exactly time $3d+4(f-3)d+3d+Cd+d$.

3.4.2 A bound for $n \leq 2f$

When $n \leq 2f$, we are able to bound the running time by an expression that depends on $\frac{f}{n-f}$. This bound requires a lemma about the length of causal sequences of r messages more complicated than Lemma 3.16. The bound is proved in the same way, using the following lemma instead of Lemma 3.16:

Lemma 3.18 *For any $r>1$, if for all $r' \leq r$ phase r' is not quiet, then $t_r - t_{r-1} \leq (3\frac{f}{n-f}+b_r+4)d$.*

Proof idea: We consider a directed graph with the processes as nodes and an edge from i to j if i sends an r message to j. Fix a path of minimal length from from a process that sends an r message by time $t_{r-1}+2d$ to a process that sends an r message to a nonfaulty process. If δ is the length of this path then by time $t_{r-1}+2d+(\delta+2)d$, all processes receive an r message. We show that $\delta \leq b_r + 3\frac{f}{n-f}$. Note that because of its minimality, each process on the path must be faulty. Therefore consider the subgraph on only *faulty* processes. The key observation is that if process i that broadcasts an r message at time $t_{r-1} + 2d$ or later, then i must *receive* r messages from all processes in A_i^r—these processes do not omit to i before sending "ack(i, r)"—and therefore has bidirectional edges to each node in A_i^r. By the minimality of our fixed path, any node ℓ at distance less than δ from the first node of the path does not send to a nonfaulty process and therefore all nodes in A_ℓ^r are contained in the subgraph. The idea is that most of the nodes at distance less than δ (all except b_r of them) have $|A^r| \geq n - f$ and so have at least $n - f - 1$ incident bidirectional edges in the subgraph. There are at most f nodes in subgraph, so δ cannot be very large; a simple argument shows that $\delta \leq \frac{3f}{n-f} + b_r$. ■

The theorem then follows in the same fashion as Theorem 3.17:

Theorem 3.19 *For $n \leq 2f$, the algorithm above solves the consensus problem for f omission failures within time $(3\frac{f}{n-f} + 5)(f + 1)d + Cd$.* ■

3.4.3 Another bound for $n \leq 2f$

When $n \leq 2f$, we are also able to bound the running time by an expression that depends on $\sqrt{C}$. In contrast to the previous two bounds, this bound does not bound the length of all phases by chains of r messages. Instead we partition the first r phases of any execution into two classes according to their length:

$$\begin{aligned} X_r &= \{i : t_i - t_{i-1} \leq \sqrt{C} \cdot d \ \text{ and } i \leq r\} \\ &= \{\text{short phases}\} \\ Y_r &= \{i : i \notin X_r \text{ and } i \leq r\} \\ &= \{\text{long phases}\}. \end{aligned}$$

Define $S_o(r) = \{p : p$ omits an r message to a nonfaulty process after time $t_{r-1}\}$. We bound the long phases by chains of r messages but bound the short phases by their defined bound. A simple argument about the length of r message chains shows the following:

Lemma 3.20 *If phase $r \geq 1$ is not quiet then either $t_r \leq t_{r-1} + (|S_o(r)| + 3)d$ or all nonfaulty processes decide by this time.* ■

The key observation is that a process cannot fail to a nonfaulty process in many long phases:

Lemma 3.21 *For any execution of the protocol taking at least ϕ phases and for any process j, there are at most $\sqrt{C} + 3$ phases $\rho \in Y_\phi$ such that $j \in S_o(\rho)$.*

Proof sketch: Process j is shut down within time $Cd + 2d \leq (\sqrt{C} + 2)\sqrt{C}d$ of when it first omits to a nonfaulty process. ■

Theorem 3.22 *For $n \leq 2f$, the algorithm above solves the consensus problem for f omission failures within time $(2\sqrt{C} + 6)(f + 1)d + Cd$.*

Proof: Let phase h be the first quiet phase. Clearly if $h = 0$ or $h = 1$, we have the desired bound. If $h > 1$, we consider two cases. Consider first the case that not all nonfaulty processes decide in phase $h - 2$. We bound the length of the short phases by their defined length. We bound the length of the long phases by Lemma 3.20 and then sum the sizes of $S_o(\rho)$ using Lemma 3.21. The length of phase h is bounded by Lemma 3.15. Thus we have $t_h - t_0 =$

$$\begin{aligned} & \sum_{\rho \in X_{h-1}} (t_\rho - t_{\rho-1}) + \sum_{\rho \in Y_{h-1}} (t_\rho - t_{\rho-1}) + (t_h - t_{h-1}) \\ \leq\ & |X_{h-1}| \cdot \sqrt{C}d + \sum_{\rho \in Y_{h-1}} (3 + |S_o(\rho)|)d + (Cd + d) \\ \leq\ & |X_{h-1}| \cdot \sqrt{C}d + 3|Y_{h-1}|d + f(\sqrt{C} + 3)d + (Cd + d) \\ \leq\ & (f+1)\sqrt{C}d + 3(f+1)d + f(\sqrt{C} + 3)d + Cd + d \\ <\ & (2\sqrt{C} + 6)(f + 1)d + Cd. \end{aligned}$$

Now consider that case that all nonfaulty processes decide in phase $h - 2$. The running time is then bounded by $t_{h-2} - t_0 =$

$$\begin{aligned} & \sum_{\rho \in X_{h-2}} (t_\rho - t_{\rho-1}) + \sum_{\rho \in Y_{h-2}} (t_\rho - t_{\rho-1}) \\ \leq\ & |X_{h-2}| \cdot \sqrt{C}d + \sum_{\rho \in Y_{h-2}} (3 + |S_o(\rho)|)d \\ \leq\ & |X_{h-2}| \cdot \sqrt{C}d + 3|Y_{h-2}|d + f(\sqrt{C} + 3)d \\ \leq\ & f\sqrt{C}d + 3fd + f\sqrt{C}d + 3fd \\ =\ & (2\sqrt{C} + 6)fd \end{aligned}$$

■

4 Byzantine failures

We now present a simulation algorithm using $3f+1$ processes and tolerating f Byzantine failures. The algorithm simulates any synchronous round-based algorithm that tolerates Byzantine failures in time $r(d+2Cd)+Cd$, where r is the number of rounds required by the synchronous algorithm.

4.1 The simulation algorithm

The simulation works by keeping processes loosely synchronized. The partial synchronization works by using a combination of two criteria for advancing to further rounds, one based on elapsed local time and the other based on messages received. A similar technique is used in [22] to initiate new rounds of clock resynchronization. Processes are synchronized only to the extent that a nonfaulty process does not advance to round r until it has received a round $r-1$ message from every nonfaulty process. We do not explore the semantics of "correct simulation", but regard this property as sufficient for simulating round-based algorithms.

Therefore, for the purposes of simulation, we define a synchronous algorithm by its message function only, suppressing information about the state of the synchronous algorithm. Let $M_i(r, V^{r-1})$ denote the vector of messages to be sent in the synchronous algorithm by process i in round r when messages V^{r-1} are received in round $r-1$ (of course, this message function may also depend on the state of the process; we leave this implicit).

Recall that we assume all processes begin executing the algorithm at the same time. At each step, a process increments a counter s (initially 0) and executes the code in Figure 3. A local variable, initially 1, keeps track of the ROUND number. Ordered set V^r contains the r^{th} message received from each process. We refer to the r^{th} message sent by a process as a "round r" message. (Assume without loss of generality that each process sends one message to all processes in every round.)

Each process first sends its round 1 message and then waits for at least time d to ensure that it receives a round 1 message from every other nonfaulty process. When it can be sure that time d has elapsed, it advances to round 2 and broadcasts its round 2 message based on the round 1 messages

```
ROUND 1    Send $M(1,\cdot)$; goto ROUND 1'.
ROUND 1'   If $s > d/c_1$ or $|V^2| \geq f+1$,
             then goto ROUND 2

ROUND r    Send $M(r, V^{r-1})$; goto ROUND $r'$.
ROUND r'   If $|V^r| \geq 2f+1$,
             then $s \leftarrow 0$; goto ROUND $r''$.
ROUND r''  If $s > (2d+3c_2)/c_1$ or $|V^{r+1}| \geq f+1$,
             then goto ROUND $r+1$.
```

Figure 3: The simulation algorithm for process i. At each step, a process increments the counter s and executes the code according to its present round number.

it has received so far. It ensures that time d has passed by either waiting for d/c_1 of its own steps or by receiving $f+1$ round 2 messages—this ensures that some nonfaulty process has waited at least time d.

In each subsequent round r, a process waits for at least time $2d$ (actually $2d+3c_2$) after at least $f+1$ *nonfaulty* processes have sent a round r message. By this time, all nonfaulty processes must have received at least $f+1$ round r messages and therefore advanced to round r and sent a round r message. At this time, a process advances to round $r+1$ and broadcasts its round $r+1$ message. Again, there are two ways for a process to deduce that sufficient time has passed: if it takes $(2d+3c_2)/c_1$ steps after receiving at least $2f+1$ round r messages or if it receives at least $f+1$ round $r+1$ messages. The latter ensures that some nonfaulty process has advanced to round $r+1$ and therefore has already waited a sufficient amount of time.

4.2 Proof of correctness

Let t_r be the latest time that any nonfaulty process sends a round r message. Again, we assume that all processes begin at the same time (here, t_1). We say a process "advances to round r" when it executes the "goto ROUND r" statement in the code. In order to prove correctness, we must show that a nonfaulty process eventually advances to all rounds required by the synchronous algorithm and always receives a round r message from all nonfaulty processes before advancing to round $r+1$.

Lemma 4.1 *Each nonfaulty process advances to all rounds required by the synchronous algorithm.*

Proof: By induction on the round number. Clearly each nonfaulty advances to round 2—it advances to round $1'$ after its first step and advances to round 2 after at most $1 + d/c_1$ more steps. For $r > 2$, each nonfaulty process receives at least $2f+1$ round r messages from the other nonfaulty processes and after at most $(2d+3c_2)/c_1$ steps in round r'', advances to round $r + 1$. ∎

Lemma 4.2 *No nonfaulty process advances to round $r+1$ before receiving a round r message from each nonfaulty process.*

Proof: By induction on the round number.

$r = 1$: Clear—each nonfaulty process takes more than d/c_1 steps, which takes more than time d, before advancing to round 2.

$r > 1$: Assuming the lemma is true for $r - 1$, we show the lemma is true for r. Let i be the first correct process to advance to round $r+1$ and let τ_i be the time at which i advances to round r''. We make the following series of deductions about the events that occur at or before the listed times:

τ_i : (By the induction hypothesis, i has received a round $r-1$ message from all nonfaulty processes.) Process i has received at least $2f+1$ round r messages

$\tau_i + d$: All nonfaulty processes are in round $(r-1)'$ or greater (because they have each sent an $r-1$ message to i) and have received at least $2f + 1$ round $r - 1$ messages (from each other).

$\tau_i + d + c_2$: All nonfaulty processes advance to round $(r-1)''$.

$\tau_i + d + 2c_2$: All nonfaulty processes have received at least $f+1$ round r messages (from the nonfaulty subset of processes that sent round r messages to i) and advance to round r.

$\tau_i + d + 3c_2$: All nonfaulty processes send a round r message and advance to round r'.

$\tau_i + 2d + 3c_2$: All processes receive a round r message from each nonfaulty process.

Process i advances to round $r+1$ only after $(2d + 3c_2)/c_1$ steps in round r'', which occurs later than time $\tau_i + 2d + 3c_2$ and hence after i has received a round r message from all nonfaulty processes. All other nonfaulty processes advance to round $r + 1$ at later times. ∎

4.3 Analysis of time bounds

Again, we assume $d \gg c_2$ and therefore approximate $d + c_2$ by d in the timing analysis.

Lemma 4.3 *$t_2 - t_1 \leq Cd$ and, for $r \geq 2$, $t_{r+1} - t_r \leq d + 2Cd$.*

Proof: Clearly, $t_2 \leq t_1 + Cd$. By time t_r all nonfaulty processes send a round r message and advanced to round r'. Therefore by time $t_r + d$, all nonfaulty processes receive at least $2f + 1$ round r messages and advance to round r''. Within another time $2Cd$, all nonfaulty processes have taken $(2d + 3c_2)/c_1$ steps and advance to round $r + 1$, sending an $r + 1$ message. ∎

Theorem 4.4 *There is an algorithm using $3f + 1$ processes which solves the consensus problem for f arbitrary failures within time $Cd + f(d + 2Cd) = fd + (2f + 1)Cd$.*

Proof: Any $(f + 1)$-round synchronous algorithm can be simulated. Agreement and validity follow from correct simulation. Termination follows from Lemma 4.1. The time bound follows from Lemma 4.3. ∎

5 Open questions

We leave open an obvious and tantalizing question: does consensus in the presence of Byzantine failures require time $\Omega(fCd)$? The difficulty of this problem seems to lie not in the potential of for arbitrary message content but in the potential for timing misbehavior. We believe an important step towards answering this question will be to obtain tight bounds for "timing failures", where the time between steps of a faulty process is not in the interval $[c_1, c_2]$.

Acknowledgments

Thanks to Cynthia Dwork, Baruch Awerbuch, Hagit Attiya, and Nancy Lynch for helpful discussions.

References

[1] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. Report TM-435, Laboratory for Computer Science, MIT, November 1990. Also in STOC 1991.

[2] H. Attiya and N. A. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. *Proc. 10th IEEE Real-Time Systems Symposium*, 1989, pp. 268–284. Also: Technical Memo MIT/LCS/TM-403, Laboratory for Computer Science, MIT, July 1989.

[3] H. Attiya and M. Mavronicolas. Efficiency of asynchronous vs. semi-synchronous networks. *the 28th annual Allerton Conference on Communication, Control and Computing,* October 1990.

[4] F. Cristian, H. Aghili, R. Strong and D. Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. *Proc. 15th Int. Conf. on Fault Tolerant Computing,* 1985, pp. 1–7. Also: IBM Research Report RJ5244, revised October 1989.

[5] B. A. Coan and C. Dwork. Simultaneity is harder than agreement. *Proc. 5th IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1986, pp. 141–150.

[6] B. Coan and G. Thomas. Agreeing on a leader in real-time. *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.

[7] R. DeMillo, N. A. Lynch and M. Merritt. Cryptographic protocols. *Proc. 14th Annual ACM Symp. on Theory of Computing,* May 1982, pp. 383–400.

[8] D. Dolev, C. Dwork and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM,* Vol. 34, No. 1 (January 1987), pp. 77–97.

[9] D. Dolev, R. Reischuk, and H. R. Strong. Eventual is earlier than immediate. *Proceedings of the 23rd IEEE Symp. on Foundations of Computer Science,* 1982, pp. 196–203.

[10] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing,* Vol. 12, No. 3 (November 1983), pp. 656–666.

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

[12] C. Dwork and Y. Moses. Knowledge and common knowledge in Byzantine environments I: crash failures. *Proc. 1st Conf. on Theoretical Aspects of Reasoning About Knowledge*, Morgan-Kaufmann, Los Altos, CA, 1986, pp. 149–170; *Information and Computation*, to appear.

[13] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters,* Vol. 14, No. 4 (June 1982), pp. 183–186.

[14] M. Fischer, N. Lynch and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM,* Vol. 32, No. 2 (1985), pp. 374–382.

[15] A. Herzberg and S. Kutten. Efficient Detection of Message Forwarding Faults. *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 1989, pp. 339–353.

[16] L. Lamport, R. Shostak and M. Pease. The Byzantine generals problem. *ACM Transaction on Prog. Lang. and Sys.*, Vol. 4, No. 3 (July 1982), pp. 382–401.

[17] M. Merritt. Notes on the Dolev-Strong lower bound for Byzantine agreement. Unpublished manuscript, 1985.

[18] M. Pease, R. Shostak and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM,* Vol. 27, No. 2 (1980), pp. 228–234.

[19] S. Ponzio. *The Real-time Cost of Timing Uncertainty: Consensus and Failure Detection.* SM thesis, Massachusetts Institute of Technology, June 1991.

[20] B. Simons, J. L. Welch and N. Lynch. An overview of clock synchronization. *Proceedings of IBM Fault-Tolerant Computing Workshop,* March, 1986.

[21] R. Strong, D. Dolev and F. Cristian. New latency bounds for atomic broadcast. *11th IEEE Real-Time Systems Symposium*, 1990.

[22] J. L. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation,* Vol. 77, No. 1 (April 1988), pp. 1–36.