THE DISTRIBUTED FIRING SQUAD PROBLEM

(Preliminary Version)

Brian A. Coan[*]
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Danny Dolev[†]
Computer Science Department
Hebrew University
Jerusalem, Israel

Cynthia Dwork[‡]
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Larry Stockmeyer
Computer Science Department
IBM Research Laboratory
San Jose, CA 95193

1. INTRODUCTION

Many fault-tolerant distributed algorithms assume a *synchronous system*, in which processing is divided into synchronous unison "steps" separated by rounds of message exchange (see, e.g., [DS, LSP]). A message sent at step s from a correct processor p to a correct processor q is received by q at step s+1. This assumption is justified by the impossibility results of [FLP] and [DDS], which show that if the system is asynchronous then there is no protocol for distributed agreement tolerant to even one benign processor failure. Another common assumption is that all processors begin the algorithm simultaneously, i.e., at the same step. In an actual distributed system in which different transactions and algorithms may be executed periodically, this may be unrealistic. Typically an algorithm is executed in response to a request from some specific processor, which may in turn be responding to some external request. If the given processor is correct then all correct processors learn of the request simultaneously, so they can indeed begin the algorithm in unison. However, if the processor is faulty then the correct processors may learn of the request at different steps. In this paper we justify the design assumption of simultaneous starts. Specifically, we provide algorithms to solve the associated synchronization problem, which we call the *distributed firing squad* problem (abbreviated *DFS*). A distributed algorithm for the DFS problem has two properties:

(1) if any correct processor receives a message to start a DFS synchronization, then at some future time all correct processors will "fire" (formally, enter a special state), and

(2) the correct processors all fire at exactly the same step.

The two complexity measures we study are *fault tolerance*, the maximum number of faulty processors which can be tolerated, and *time*, the maximum number of rounds of message exchange taken by the algorithm starting with the step at which some correct processor receives a message to start a DFS synchronization and ending with the step at which all correct processors fire. We are also interested in the *communication complexity* of an algorithm, that is, the total number of message bits exchanged, but only to the extent of distiguishing polynomial from exponential communication complexity. Below, *n denotes the number of processors in the system; t* denotes the maximum number of faults that can be tolerated by a particular algorithm, and any such algorithm is said to be *t-resilient*.

In the case of fail-stop faults (the most benign type of fault usually studied, in which a faulty processor follows its

[*]Work supported by the NSF under Grant #DCR-8302391 and by the U. S. Army Research Office under Contracts #DAAG29-79-C-0155 and #DAAG29-84-K-0058, and by the Advanced Research Projects Agency of the Department of Defense under Contract #N00014-83-K-0125.

[†]Batsheva de Rothschild Fellow. Work performed in part while visiting IBM, San Jose.

[‡]Bantrell Postdoctoral Fellow. Work performed in part while visiting IBM, San Jose.

algorithm correctly but simply stops at some point), it is easy to find a t-resilient DFS algorithm for any number $t \leq n$ of faults which halts in t+1 rounds. This was observed independently by James Burns and Nancy Lynch. By reducing the Weak Byzantine Agreement (WBA) problem to the DFS problem, we can use a lower bound of Lamport and Fischer [LF] on the time complexity of the WBA problem to show that any t-resilient algorithm for the DFS problem requires t+1 rounds for fail-stop faults, and therefore also for more serious types of faults. Thus, the situation for fail-stop faults is well understood. Burns and Lynch [BL] give a DFS algorithm for the case of Byzantine faults without authentication (the most serious type of fault usually studied, where faulty processors can exhibit arbitrary behavior); we say more about this case below. The main results in this paper concern Byzantine faults with authentication. Byzantine processors can exhibit arbitrary behavior, but we assume that every processor can sign messages in such a way that the signature of a correct processor cannot be forged by a faulty processor (see, e.g., [DS]).

In trying to determine the maximum fault tolerance of the DFS problem in the authenticated Byzantine case, we found it necessary to distinguish between several types of faulty behavior. In *rushing* a Byzantine faulty processor can receive, process and re-send messages "between" the synchronous steps of other processors. Figure 1(a) shows a normal communication round involving three correct processors A, B and C, with A sending messages to B and C. Figure 1(b) shows a similar round in which processor C is faulty, takes a step between the steps of the correct A and B, computes its response to the message it received from A, and then "rushes" this response to B in the same round. A special case of rushing is the *timing fault* model where faulty processors never fail and always follow their algorithms correctly, but may take steps at irregular times and may experience slight delays or accelerations in communicating with the other processors. Rushing and timing faults are realistic types of faults whenever there is sufficient uncertainty in message transmission time. The length of a communication round must be chosen as large as the maximum possible transmission time between correct processors, but if a message happens to be delivered to a faulty processor in time less than this maximum, the faulty processor has the opportunity to rush.

We must also distinguish the case where faulty processors can sign messages using the encryption functions of other faulty processors, which we call *collusion*, and the case where a faulty processor has only its own encryption function. Collusion is unlikely to occur as a result of a random failure, but it could occur if the faulty processors were controlled by a malevolent intelligence which allowed faulty processors to share encryption keys.

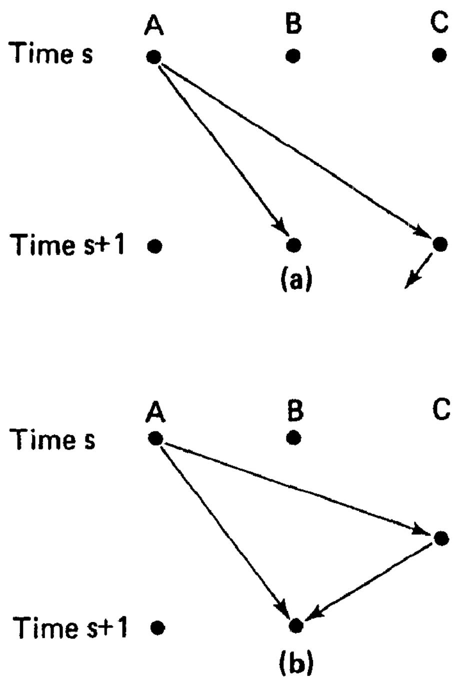

Figure 1. (a) A communication round with three correct processors. For simplicity, only the messages sent by A are shown.
(b) The processor C rushes.

Table 1 summarizes our results and the results of [BL] for the different fault models. Each entry gives n_{min}, the smallest number n ($n \geq 2$) of processors for which there exists a t-resilient distributed firing squad algorithm ($t \geq 1$). Unless otherwise indicated, all algorithms require at most t+c rounds where $c \leq 5$ is a constant independent of n and t, and the total number of bits of communication is polynomial in n. In proving lower bounds on the minimum n, we make the usual assumption that the receiver of a message knows the identity of the sender; however, this assumption is not needed by our algorithms (upper bounds on n_{min}).

There are several interesting things to note about these results. First we should emphasize that the lower bound $n \geq 3t+1$ holds for the timing fault model in which *all processors follow the algorithm correctly*. The only way faulty processors can affect the system is by taking steps at irregular times and unknowingly delaying and speeding up certain messages by small amounts. Secondly, even though our bounds on the minimum n in the case of collusion but no rushing are presently not tight, the bounds are sufficient to show that collusion does decrease fault tolerance when compared to the case of no collusion and no rushing, and rushing alone admits less fault tolerance than collusion alone. The

Fault	Minimum n for t-resiliency	Remarks
fail-stop	t	Due indep. to Burns and Lynch.
Byzantine with authentication		
no rushing, no collusion	t	Due indep. to Burns and Lynch.
collusion, no rushing	$5t/3 < n_{min}$	Lower bound valid only for $t \geq 3$.
	$n_{min} \leq 2t+1$	Algorithm takes $2t+1$ rounds.
rushing, no collusion	$3t+1$	
rushing and collusion	$3t+1$	
timing faults	$3t+1$	
Byzantine without authentication	$3t+1$	Algorithm due to Burns and Lynch [BL]. Algorithm uses exponential communication or more than $t+O(1)$ rounds.

Table 1

distinction thus shown between these three fault models is (to us) one of the unexpected results of this work.

Burns and Lynch [BL] solve the DFS problem in the nonauthenticated Byzantine case essentially by adapting an agreement protocol. Since all known nonauthenticated agreement protocols either use exponential communication, use more than $t+O(1)$ rounds, or require $n > t^2$ [DFFLS, DRS, LSP], their DFS solution has the same property. By using signatures, we are able to achieve polynomial communication, $t+O(1)$ time, and the maximum resiliency $\lfloor (n-1)/3 \rfloor$ simultaneously. Our lower bounds $n > 3t$ ($n > 5t/3$) for rushing (collusion) suggest that the approach of directly adapting agreement protocols to the DFS problem will not work in the authenticated case, since there are authenticated agreement protocols tolerant to any number of failures [DS]. For the same reason, the DFS problem seems to be different than the clock synchronization problem studied in [HSSD, LM, LL] where the object is to bring the clocks of correct processors "close" together. In the authenticated Byzantine case, there is a clock synchronization algorithm tolerant to any number of failures [HSSD]. Our problem is also significantly different than the version of the firing squad problem which was proposed in the late 1950's [Mo]. That version of the problem was interesting since the processors were finite state machines which were connected in a linear array so each processor could only count to some fixed constant independent of n; however, faults were not considered. In our version of the problem, the difficulty arises not from limitations on the processors or communication network (since we assume a completely connected system of powerful processors) but rather from the possibility of Byzantine faulty processors and timing faults.

Apart from the motivation discussed at the beginning of this section, firing squad synchronization seems to be a basic problem in distributed computing. Much of the theoretical work on fault-tolerant distributed computing has concentrated on the agreement problem. More recently, attention has shifted in part to other basic problems such as clock synchronization [DHS, HSSD, LM, LL] and distributed coin flipping [BD]. Through this paper, we hope to add firing squad synchronization to the list of basic problems in distributed computing which are amenable to theoretical understanding.

In Section 2 we give definitions. Section 3 contains results (DFS algorithm and lower bound) for the case of no collusion and no rushing, Section 4 gives results for rushing and timing faults, and Section 5 considers the case of collusion but no rushing. In Section 6 we mention some related results, such as the application of DFS ideas to the problem of Byzantine Agreement in the case that the processors do not all start at the same round.

2. DEFINITIONS

For simplicity we give definitions for a single occurrence of a firing squad synchronization. Any algorithm which solves a single occurrence can be easily modified to solve several occurrences in sequence or even concurrently. Let p_1, p_2, ..., p_n denote the processors in the system. For technical reasons we introduce another "processor" w whose only purpose is to start a DFS synchronization; w does not receive messages from the p_i's. In reality w might be another process running within one of the p_i. Formally a DFS algorithm is specified by an infinite set of messages M and for each processor p_i an infinite set of states Q_i, a state transition function σ_i, and a sending function β_i, where

$$\sigma_i: Q_i \times M^{n+1} \rightarrow Q_i$$
$$\beta_i: Q_i \times M^{n+1} \rightarrow M^n.$$

The inputs to σ_i and β_i are the current state and an (n+1)-tuple of received messages, one from each processor p_1, ..., p_n, w. σ_i gives the new state and β_i gives an n-tuple of messages $(m_1,...,m_n)$ such that m_j is sent to p_j for each j. There are special messages ϕ, the null message, and "Awake", the awake message. For each i there are states q_0, q_f in Q_i, the *quiescent state* and the *firing state*, respectively. In addition,

$$\sigma_i(q_0, \phi, ..., \phi) = q_0,$$
$$\beta_i(q_0, \phi, ..., \phi) = (\phi, ..., \phi).$$

We assume that processors take steps at times specified by nonnegative real numbers. A *run* is specified by giving, for each processor p_1, ..., p_n, w, a list of nonnegative real numbers which specifies the times at which the processor takes steps. A message sent from a processor p to a processor q at time s is received by q at time s' where s' is the smallest $s' > s$ such that q takes a step at s'. (If q receives more than one message from some p at some step, then the message sent at the latest time is used by the transition functions; since this occurs only if either p or q is faulty, this convention is not critical.) Whenever w takes a step it sends the awake message to some (possibly empty) subset of the p_i's and it sends the null message to the rest. The concept of global time is introduced just as an expositional convenience. The individual processors have no knowledge of global time.

Definition. A processor p_i is *correct* in a run R if

(1) p_i takes its first step at time 0 in state q_0 receiving messages $(\phi, \phi, ..., \phi)$, and thereafter takes steps at successive integer times 1, 2, 3, ..., and

(2) p_i executes its algorithm (transition functions) correctly.

A run R is *active* if some correct processor receives a non-null message at some step; define awake(R) to be the earliest such time (necessarily integral). If a correct p_i receives a non-null message for the first time at time s, we say that p_i *awakens* at time s. Define $\text{fire}_i(R)$ to be the time of the first step in R during which p_i makes a transition into state q_f (undefined if p_i does not enter q_f).

Definition. A DFS algorithm is *t-resilient* with respect to a given type of faulty behavior if for any active run R in which at most t of the processors p_1, ..., p_n are faulty and in which the faulty processors conform to the given type of faulty behavior, there is a (necessarily integral) time fire(R) $\geq$ awake(R) such that $\text{fire}_i(R) = \text{fire}(R)$ for all i such that p_i is correct in R. The *time complexity* of the DFS algorithm is the maximum of fire(R) − awake(R) over all such runs R. (Note that w is not counted among the t faulty processors no matter how it behaves.)

We now define various types of faulty behavior. A faulty processor p_i is *fail-stop* if it operates as a correct processor up to some step s, at step s some (possibly empty) subset of the messages p_i is supposed to send are replaced by null messages, and for all subsequent steps p_i sends only null messages. The definitions of authenticated and nonauthenticated Byzantine faultiness are well known in the literature [DS, LSP], the definitions are reviewed in the Introduction, and they are not repeated here. A faulty processor *rushes* in a run R if it takes some step at a non-integer time in R. In this model of failure, messages to and from faulty processors may take less than one round to be delivered. A faulty processor *colludes* in a run R if it signs a message using the signature function of another processor which is also faulty in R.

E_p denotes the signature function of processor p.

Finally we define the *timing fault model*. Runs in this model have the following properties:

(1) all processors execute the algorithm correctly,
(2) correct processors take steps at times 0, 1, 2, ...,
(3) faulty processors take steps at times 1/2, 3/2, 5/2, ...,
(4) messages between two correct processors or between two faulty processors take time 1 to be delivered,
(5) messages between a correct and a faulty processor take either time 1/2 or time 3/2 to be delivered.

It is not hard to see that the timing fault model is a special case of authenticated Byzantine faultiness with rushing (but no collusion). The model with rushing can simulate a delivery time of 3/2 simply by having a Byzantine processor either delay sending or delay receiving the message. For example, if in the timing fault model the faulty processor p sends a message m at time 3/2 which the correct q should receive at time 3, then in the model with rushing the (now Byzantine) p simply holds m and sends it to q at time 5/2. Therefore, giving a DFS algorithm for the model with rushing yields a result for both models, as does proving a lower bound on n for the timing fault model.

3. NO RUSHING AND NO COLLUSION

We begin with a simple algorithm which tolerates any number of fail-stop or authenticated Byzantine faults. It does not tolerate rushing, timing faults, or collusion. This algorithm was discovered independently by Burns and Lynch. The basic idea is that since any processor, faulty or otherwise, can add at most one signature per round, we can use the number of signatures on a message as a clock, giving a lower bound on the time elapsed since the protocol was initiated. A correct processor fires as soon as it knows that at least t+1 rounds have elapsed. The details of the algorithm and its proof of correctness are similar to the Dolev-Strong algorithm for authenticated Byzantine agreement [DS]. Details will be given in the final version of the paper.

Theorem 3.1. In the model with authenticated Byzantine failures (but no rushing or collusion) there is a t-resilient DFS algorithm for any number $n \geq t$ of processors. The algorithm has time complexity of t+1 rounds and it uses an amount of communication polynomial in n.

Obviously, the algorithm of Theorem 3.1 works also for fail-stop faults. We now show that the time complexity of this algorithm is optimal by reducing the Weak Byzantine Agreement problem (WBA) [L] to the distributed firing squad problem. Optimality follows from the fact that WBA requires at least t+1 rounds [LF].

In the WBA problem, all processors start the algorithm at the same global time (say, time 0) and each processor has a binary initial value. In particular, by maintaining a counter all correct processors have a common notion of global time. A protocol solves WBA if (1) every correct processor eventually reaches a decision, (2) no two correct processors reach different decisions, and (3) if all initial values are the same, say v, and there are no failures, then v is the value decided. The following result is due to Nancy Lynch and is a slight modification of a reduction which we found.

Theorem 3.2. Let A be an algorithm for DFS which is t-resilient to fail-stop faults (resp., unauthenticated Byzantine faults) and which requires k rounds between awakening and firing in the execution in which all the processors awaken simultaneously and no failure occurs (note that k is unique since the system is completely deterministic in this case). Then there exists an algorithm for WBA which is t-resilient to fail-stop faults (resp., unauthenticated Byzantine faults) and which *always* halts in k rounds.

Proof. Consider an instance of WBA in which processor p_i has initial value v_i. If $v_i = 0$ then p_i begins simulating A at time 0. That is, p_i acts as if it received the awake message from w and null messages from the rest. If $v_i = 1$, then p_i begins simulating A at time 1. That is, p_i sends null messages during the first round and acts as though it received the awake message from w at time 1 (p_i could receive non-null messages from other processors at time 1 in this case if other processors had initial value 0). If the simulation of A causes p_i to fire at time k or earlier, then p_i decides 0 at time k; otherwise pi decides 1 at time k.

Correctness of A immediately implies that all correct processors decide on the same value, since either all correct processors simulate a firing at a time $\leq$ k or none do. If all processors begin with value 0 and there are no failures, then by choice of k each processor will simulate a firing at time k, so the decision will be 0. However, if all begin with 1 and there are no failures, then all processors will simulate a firing at time k+1, so the decision will be 1. □

Corollary 3.3.

(1) Any protocol for DFS resilient to t fail-stop faults requires at least t+1 rounds. Moreover, this is true even if the order in which processors are sent to in a broadcast is fixed *a priori*. It is also true even in some execution in which all processors are correct.

(2) Any protocol for DFS resilient to t unauthenticated Byzantine failures requires at least 3t+1 processors.

Proof. The proof is immediate from the preceding theorem and the corresponding bounds for WBA [L, LF, FLM, Me]. □

In the next section the lower bound of Corollary 3.3(2) is strengthened to the model with authentication and rushing.

4. RUSHING AND TIMING FAULTS

This section contains tight bounds for the two new models: timing faults and Byzantine failures with rushing. The following result gives the principal algorithm of the paper.

Theorem 4.1. In the model with Byzantine failures and authentication, in which faulty processors can both rush and collude, there is a t-resilient distributed firing squad protocol requiring t+5 rounds, $n \geq 3t+1$ processors, and communication polynomial in n.

We begin with an informal discussion of the principal ideas of the algorithm. Our protocol is composed of a set of indentical subprotocols executed independently and in parallel. A processor initiates a subprotocol by broadcasting its signature. Let p be an arbitrary (possibly faulty) initiator, and consider a set of processors all receiving p's signature at the same step. In some sense these processors are synchronized, in that they share a common idea of when they first heard from p, although no processor in the set knows which other processors are in the set. If the set of synchronized processors is sufficiently large, then *because they are synchronized* these processors can run an agreement protocol similar to the Dolev and Strong protocol [DS] which assumes synchronous start. The processors are essentially agreeing on the members of the set. If the agreed upon set is sufficiently large, then correct processors will order a firing. In particular, a correct processor in the set only orders a firing if there are at least $n-t \geq 2t+1$ processors in the agreed upon set. Of these, at least $t+1$ are correct, synchronized processors. Thus a correct processor only orders a firing if at least $t+1$ correct processors do so simultaneously. Now, consider a processor q receiving at least $t+1$ commands to fire. Since q knows at least one of these messages is from a correct processor it knows at least $t+1$ are. Thus q knows that every processor receives at least $t+1$ commands to fire, and therefore that every processor knows every processor has received these commands and so on. In short, it becomes *common knowledge* that every processor has received $t+1$ commands to fire, so it is safe to fire.

Proof of Theorem 4.1. As stated above, the protocol is composed of a set of identical subprotocols executed independently and in parallel. Specifically, as each correct processor awakens it initiates a *core protocol*. If the core protocol is successfully completed then the correct processors fire upon completion. A core protocol initiated by a correct processor will complete successfully unless a firing occurs earlier due to the completion of a different execution of the core protocol. A core protocol initiated by a faulty processor may not cause a firing, but if it does then all correct processors fire simultaneously. (Thus, it would be sufficient to have any $t+1$ processors initiate the core protocol.) In the following, if a correct processor receives the same message at different times, all receptions but the first are ignored; this prevents a faulty processor from doing any damage by taking a message which was broadcast by a correct processor and resending it at a later time.

A processor p initiates a core protocol by broadcasting its signature, $E_p(p)$. Each processor (including p itself) which receives $E_p(p)$ signs it and broadcasts it. Each processor q then attempts to form a *core for p*, which is a list of the form

$$<E_{i_1}(E_p(p)), \ldots, E_{i_k}(E_p(p))>$$

where $k \geq n-t$ and each of the k copies of $E_p(p)$ is signed by a distinct processor. The signatures $E_{i_1}, \ldots, E_{i_k}$ *belong* to the core, and the core *contains* these signatures. Intuitively, a core is a set of processors which claim to have received $E_p(p)$ at the same step. For technical reasons, we found it necessary to *notarize* the core.

A *notarized core for p* is a list

$$<E_{i_1}(C_1), \ldots, E_{i_k}(C_k)>,$$

where $k \geq n-t$ and the C's are (possibly different) cores for p, each signed by a distinct processor.

A processor q which receives $E_p(p)$ at time s tries to form a core for p at time $s+1$. This is the only time at which q tries to form a core for p, and q only includes in the core messages received at time $s+1$. If q forms a core then q includes in the core all messages of the form signature($E_p(p)$) received at time $s+1$. If a core is formed, q signs it and broadcasts it. Processor q also tries to form a notarized core for p at time $s+2$. A notarized core, if formed, contains all messages of the form signature(core for p) received at time $s+2$. If a notarized core N is formed by q at this step, then N is considered to have been "received" at this step. Starting with the second step after $E_p(p)$ was received, each processor q does the following (this is done regardless of whether or not q formed a core for p or a notarized core for p).

If q receives message m, q checks if m is *acceptable* in the following sense:

(1) $m = E_{j_1}(E_{j_2}(\ldots(E_{j_k}(N))\ldots))$ where N is a notarized core for p and each of the k signatures ($k \geq 0$) are distinct (m is said to have *length* k, denoted $|m|$),

(2) q's signature belongs to at least $n-2t$ of the cores in N (we say that q *supports* N), and

(3) q first received $E_p(p)$ $k+2$ steps ago (this condition implies that at any given step of q, messages of only one particular length are acceptable).

An acceptable message m as in (1) is *new* to q if none of the signatures $E_{j_1}, E_{j_2}, \ldots, E_{j_k}$ are by q. If q finds one or more messages of length k which are new and acceptable, q chooses one such message m arbitrarily and broadcasts $E_q(m)$, ignoring the rest. Finally, if q receives an acceptable message m of length $t+1$, then q signs and broadcasts "fire_p". A correct processor fires at step f if and only if at step f it receives at least $t+1$ commands "fire_p" signed by different processors.

Lemmas 4.1.1-4.1.4 show that the core protocol causes a firing if the initiator is correct. For these lemmas, let p be a correct processor initiating a core protocol at time r.

Lemma 4.1.1. At time $r+2$ all correct processors can form a core for p containing the signature of every correct processor, and at time $r+3$ all correct processors can form a notarized core for p.

Proof. Since p is correct all correct processors receive $E_p(p)$ at time $r+1$. All correct processors q broadcast $E_q(E_p(p))$ at time $r+1$, and these messages are received at time $r+2$. Since there are at least $n-t$ correct processors every processor receives at least $n-t$ messages of the form $E_q(E_p(p))$ signed by distinct processors. Thus all correct processors can form a core at time $r+2$. Further, since a correct processor puts all messages $E_i(E_p(p))$ received into the core, for every correct processor q the message $E_q(E_p(p))$ appears in the cores formed by the correct processors.

A similar argument shows that every correct processor can form a notarized core at time $r+3$. □

Lemma 4.1.2. Let N be any notarized core for p. Then every correct processor q supports N.

Proof. A notarized core contains at least $n-t$ cores, each signed by distinct processors, so at least $n-2t$ of the cores contained in N were formed by correct processors. By Lemma 4.1.1 all these $n-2t$ cores contain the signature of every correct processor. □

Lemma 4.1.3. For all i, $0 \leq i \leq t$, at time $r+3+i$ at least one correct processor receives a new acceptable message (of length i).

Proof. The proof is by induction on i.

Basis $i = 0$. By the previous two lemmas every correct processor forms a notarized core which it supports at time $r+3$. By convention, this notarized core is a new acceptable message "received" at time $r+3$.

Assume the lemma true inductively for $i-1$ ($i \geq 1$). Thus at time $r+3+(i-1) = r+i+2$ some correct processor receives a new acceptable message of length $i-1$. It signs this message and broadcasts the resulting message m of length i with notarized core N. The message m is received at time $r+i+3$ by all correct processors. Since there are $n-t \geq 2t+1$ correct processors, at most t of which have signed m, and since by Lemma 4.1.2 every correct processor supports N, m is acceptable to some correct processor that has not yet signed it, so the induction holds. □

Lemma 4.1.4. If a correct processor p initiates a core protocol at time r then the core protocol runs to completion and the correct processors fire at time $r+t+5$.

Proof. By Lemma 4.1.3, at time $r+3+t$ at least one correct processor receives a new acceptable message m. Thus by time $r+4+t$ every correct processor receives an acceptable message m of length $t+1$, so all correct processors broadcast the command to fire. Since there are at least $n-t > t+1$ correct processors, every processor receives at least $t+1$ commands to fire at time $r+5+t$, so a firing will indeed take place at time $r+5+t$. □

We now show that for an arbitrary initiator p, the core protocol never causes two correct processors to fire at different times. Let p be a possibly faulty processor initiating a core protocol. If S is a set of processors, we say that S *forms a core for* p if any processor in S forms a core for p. A *group* is a maximal set of correct processors receiving $E_p(p)$ at the same time. Let G be a group and let s be the time when G receives $E_p(p)$.

Let H be the set of correct processors not in G.

Lemma 4.1.5. If G forms a core for p then H does not form a core for p.

Proof. First we observe that if G forms a core then the core contains no signatures of processors in H. Similarly, no signature of a processor in G is contained in a core formed by any processor in H.

If G forms a core for p then there exists some g in G that received at least $n-t$ messages of the form $E_q(E_p(p))$ at time $s+1$. Since none of those messages were sent by processors in H we have $|H| \leq t$. Thus even if the processors in H form a group and t faulty processors cooperate in helping H to form a core, the total number of cooperating processors is $2t < n-t$, so H cannot form a core. □

Lemma 4.1.6. If G forms a core for p and if any processor forms a notarized core N for p, then every processor in G supports N.

Proof. Every notarized core N contains at least $n-t$ cores, at least $n-2t$ of which were formed by correct processors. Since no processor in H forms a core at least $n-2t$ of the cores in N were formed by processors in G and therefore contain all the signatures of all the processors in G. □

Lemma 4.1.7. Let N be a notarized core for p. If some g in G supports N then:

(1) all processors in G support N, and

(2) no processor in H supports N.

Proof. We will show that if g belongs to $n-2t$ of the cores in N then G forms a core for p. It follows by Lemma 4.1.6 that every processor in G supports N. This will give us (1). Further, by Lemma 4.1.5 if G forms a core then H does not. Since neither processors in G nor in H form cores containing signatures of processors in H, the only cores which contain processors in H are formed by faulty processors. Thus, there can be at most $t < n-2t$ of them, so we have (2).

It remains to show that if g belongs to at least $n-2t > t$ of the cores in N then G forms a core. This is immediate from the fact that no processor in H forms a core containing elements of G. Thus if g appears in more than t cores, at least one of these was formed by some processor in G. □

Lemma 4.1.8. If any processor in G ever finds a message acceptable then G contains at least $n-2t$ processors.

Proof. Let m be acceptable to some g in G and let N be the notarized core of m. Of the $n-2t \geq t+1$ cores in N containing g at least one is signed by a correct processor. Let q be such a correct processor and let C be the core in N signed by q; i.e., $E_q(C)$ has the form

$$E_q(C) = E_q(< \ldots, E_g(E_p(p)), \ldots >).$$

Of the $n-t$ processors whose signatures belong to C, at least $n-2t$ are correct. These $n-2t$ correct processors (one of which is g) all wrote to q at the same time indicating that they received $E_p(p)$ at that time. Since no processor in H received $E_p(p)$ at the same time as g, no processor in H belongs to C. Since the correct processors are in either G or H, it follows that G contains at least $n-2t$ processors. □

Lemma 4.1.9. Let m be a message which is new and acceptable to processor g in group G at time z. Then $E_g(m)$ is acceptable to all processors in G at time z+1.

Proof. Let N be the notarized core of m. One of the conditions of acceptability is that g supports N. By Lemma 4.1.7, every processor in G supports N. By condition (3) of acceptability, g first received $E_p(p)$ at time $z - |m| - 2$, as did all other processors in G (by definition of a group), so every processor in G first received $E_p(p)$ at time $(z+1) - |E_g(m)| - 2$. Thus every processor in G finds $E_g(m)$ acceptable at time z+1. □

Lemma 4.1.10. Let f be the earliest time at which some correct processor q fires (as a result of the core protocol initiated by p). Then all correct processors fire at time f.

Proof. Since q fires only if it simultaneously receives at least t+1 messages "fire_p", some correct processor g sent "fire_p" at time $f-1$. Therefore, g received an acceptable message m of length t+1 at time $f-1$. Let G be the group of g. Without loss of generality let

$$m = E_{t+1}(E_t(...(E_1(N))...)).$$

Let $c = p_j$ be a correct processor among the t+1 processors that signed N. Let

$$m' = E_{j-1}(...(E_1(N))...).$$

Since c finds m′ acceptable, c supports N. Since g finds m acceptable, g supports N. It follows from Lemma 4.1.7(2) that c belongs to G. Let z be the time when c receives m′. By Lemma 4.1.9, all processors in G find $E_c(m')$ acceptable at time z+1. Further, by Lemma 4.1.8, G contains at least $n-2t \geq t+1$ processors, so there will be some processor in G which has not yet signed N, provided $|E_c(m')| \leq t$. By repeated application of Lemmas 4.1.9 and 4.1.8, all processors in G receive an acceptable message of length t+1 at time $f-1$, so they all broadcast fire_p at time $f-1$. Recall that G contains at least t+1 processors. It follows from the definition of the core protocol that all correct processors fire at time f. □

The proof of Theorem 4.1 follows directly from Lemmas 4.1.4 and 4.1.10. It is clear from the definition of the protocol that the number of bits of communication is polynomial in n.

We next give a matching lower bound, $n \geq 3t+1$, for the timing fault model. As noted in Section 2, the lower bound of Theorem 4.2 holds also for the fault model of Theorem 4.1 (even without collusion).

Theorem 4.2. In the timing fault model there is a t-resilient DFS algorithm only if $n \geq 3t+1$.

Proof. Consider first the proof that there is no DFS algorithm for $t = 1$ and $n = 3$. We consider four scenarios with three processors, A, B and C, in each. Processor C is faulty in Scenarios 1 and 4, B is faulty in Scenario 2, and A is faulty in Scenario 3. It is possible to fix the wake-up times and the message transmission times (see Figure 2) so that the following facts hold.

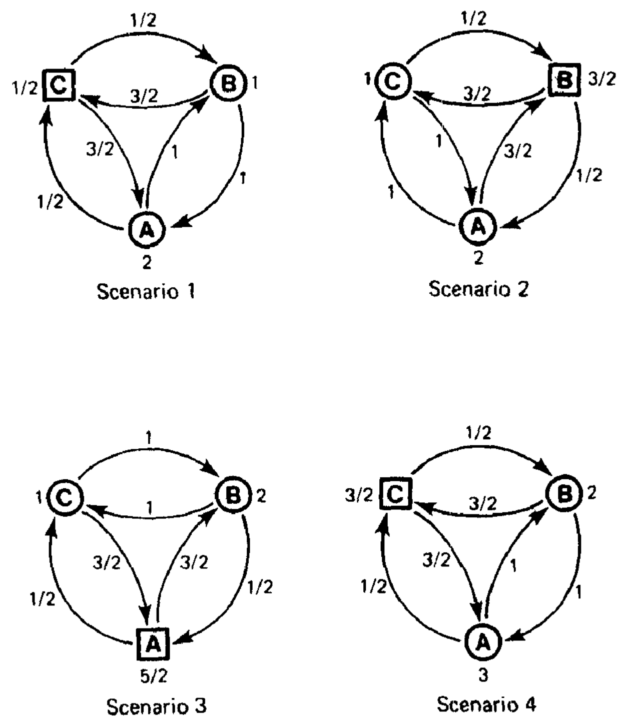

Figure 2. The Scenarios used to prove Theorem 4.2. The number on the edge directed from processor X to processor Y is the message transmission time from X to Y. The number written next to processor X is the time when processor X wakes up. Correct (faulty) processors are drawn as circles (squares).

Lemma 4.2.1. If A fires at time z in Scenario 1 then A fires at time z+1 in Scenario 4.

Proof. This follows since Scenarios 1 and 4 are identical except that all processors wake up exactly one time unit later in Scenario 4. □

For the next lemma it is convenient to introduce the "local step number" of a processor. A processor executes its first local step at the time it wakes up, and the local step number is incremented by one at each subsequent step. For example, in Scenario 1 in Figure 2, A is executing its first step at global time 2, whereas B is executing its second step at global time 2. Letting p denote a processor, two scenarios are *p-equivalent* if the message history of p, i.e., messages received and messages sent at each local step of p, are the same in the two scenarios. Two scenarios are *strongly p-equivalent* if they are p-equivalent and p wakes up at the same global time in both scenarios.

Lemma 4.2.2. For i = 1,2,3, Scenarios i and i+1 are strongly p-equivalent where p is the processor that is correct in both scenarios.

Proof. By inspection of the scenarios in Figure 2, one can easily verify that the following two facts hold for all four scenarios and for all integers $s \geq 1$:

(1) for all messages sent from A to B, from B to C, or from A to C, the message sent at local step s of the sender is received at local step s+2 of the receiver, and

(2) for all messages sent from B to A, from C to B, or from C to A, the message sent at local step s of the sender is received at local step s of the receiver.

It follows easily from these facts (formally by induction on the local step number) that any two scenarios are p-equivalent where p is any of the three processors. The lemma then follows immediately from the choice of the wake-up times. □

These lemmas easily give a contradiction. Say that A fires at time z in Scenario 1. By strong A-equivalence of Scenarios 1 and 2, A fires at time z in Scenario 2. Since A and C are correct in Scenario 2, C also fires at z in Scenario 2. By a similar argument, B fires at z in Scenario 3, and A fires at z in Scenario 4, which contradicts Lemma 4.2.1.

The impossibility proof for general n and t with $n \leq 3t$ is done as usual by replacing each processor by a group of at least one and at most t processors. The intragroup transmission times are all 1. The intergroup transmission times and the wake-up times are chosen as in Figure 2. This completes the proof of Theorem 4.2. □

5. COLLUSION

In this section we examine the distributed firing squad problem in the authenticated Byzantine model, in which faulty processors may share signature functions but they cannot rush messages.

Theorem 5.1. In the model with Byzantine failures and authentication where faulty processors can collude but cannot rush, there exists a t-resilient algorithm for the DFS problem requiring $n \geq 2t+1$ processors, 2t+1 rounds, and an amount of communication polynomial in n.

Proof sketch. As in the protocols for the models without rushing and with more benign types of failures, correct processors attempt to build messages signed by several processors and to use the length of these messages to synchronize. Since faulty processors can add several signatures at a given step we wish to obtain a sort of "notarization" for each signature in a string of signatures guaranteeing that a specific amount of time was spent adding the signature. In the straightforward approach, a processor p requests notarization of a signed message $E_p(m)$ by broadcasting $E_p(m)$. Then all processors attempt to obtain at least t+1 acknowledgements of the form $E_q(E_p(m))$. The list

$$m' = <E_{q_1}(E_p(m)), \ldots, E_{q_{t+1}}(E_p(m))>$$

is the notarization of $E_p(m)$. If the length of a message is the number of notarizations it has undergone, then a message of length k requires exactly 2k steps to be constructed, even if the k signers of the message are faulty. Although conceptually simple this approach leads to an algorithm with communication complexity exponential in t.

Our algorithm uses the idea of notarization with an implementation which is harder to prove correct but which requires communication only polynomial in n and t. Briefly, a processor p *requests support* for a message $E_p(m)$ by broadcasting $<E_p(m), P(m)>$ where P(m) is a proof that t+1 processors support m, i.e., a list of t+1 messages of the form "support m" signed by different processors. (If m contains no signatures then no proof is required.) Any processor receiving such a message supports it by broadcasting "support $E_p(m)$". The key observation is that the proof P(m) can be thrown away at this point. If some other processor q which has not yet signed $E_p(m)$ can form a proof of support for $E_p(m)$, then q requests support for $E_q(E_p(m))$ by broadcasting $<E_q(E_p(m)), P(E_p(m))>$, and so on. A processor fires if it receives a request message of the form $<E_p(m), P(m)>$ where $E_p(m)$ contains t+1 distinct signatures. Viewing the number of signatures on a message as a clock, the two key lemmas state that (1) the correct processors can increment the clock by 1 within two steps, and (2) the faulty processors cannot increment the clock faster than this. The idea of notarization and its efficient implementation is similar to the fault-tolerant distributed clocks described in [ADG, DLS]. Details will appear in the final version of the paper. □

Theorem 5.2. In the fault model of Theorem 5.1, if $t \geq 3$ there is a t-resilient DFS algorithm only if $n \geq \lfloor 5t/3 \rfloor + 1$.

Proof. The general outline of the proof is similar to the proof of Theorem 4.2. Consider the impossibility proof for t = 3 and n = 5. We consider six scenarios, with three faulty and two correct processors in each. Figure 3 shows the message transmission times, wake-up times, and which processors are faulty in each scenario. A link which is not drawn in these scenarios means that the faulty processor at one end of the link does not communicate along that link; i.e., no messages

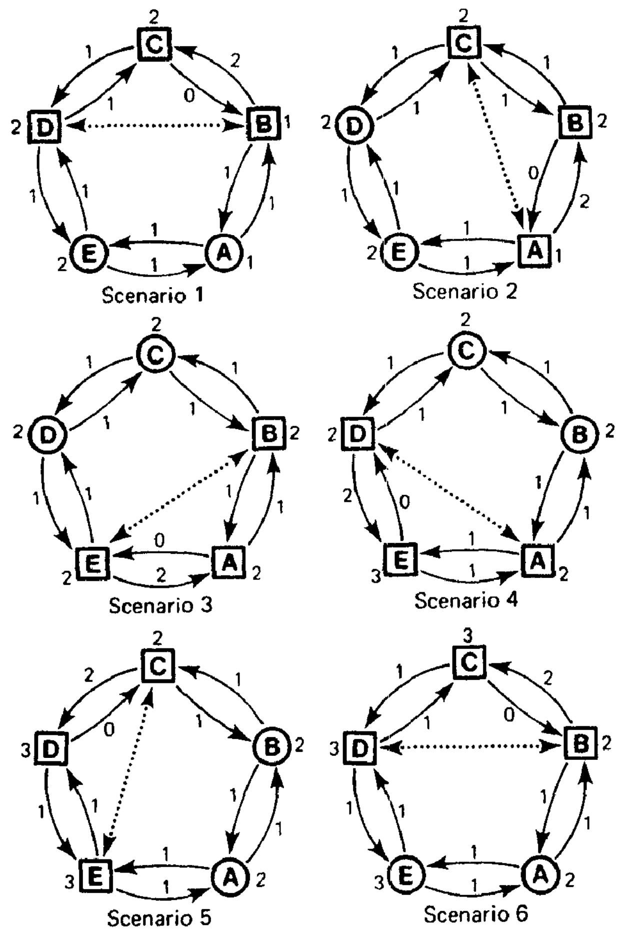

Figure 3. The Scenarios used to prove Theorem 5.2.

are sent along that link by the faulty processor and messages received along that link are ignored. The link drawn as a dotted line is used only by faulty processors. Therefore, in each scenario the network is essentially a ring from the point of view of a correct processor. In each scenario, two of the faulty processors simulate a "timing fault" where messages in one direction take time 2 and messages in the other direction take time 0. The only nonobvious part is simulating a transmission time of zero. To see how this is done focus, for example, on Scenario 1 where messages from C to B take zero time. Note that D is also faulty in Scenario 1. Whenever D takes a step at some time x in which it should send the message m to C, it sends m to B also. At time x+1, B has enough information to do the processing that C would do at time x+1 to find the message m' that C should send to B at time x+1 (the ability of B to sign messages with C's signature is necessary here). But B has m' during the step it is executing at time x+1, thus simulating the transmission of m' from C to B in zero time. Message transmission time of 0 is simulated similarly in the other scenarios.

By following the proof of Theorem 4.2, it is straightforward to show that analogues to Lemmas 4.2.1 and 4.2.2 hold. (Formally, in defining equivalence of scenarios, only messages sent along the ring links are included in message histories; the messages sent over dotted links are not included.)

The proof for general n and t is done by replacing each processor by a group of at most $\lfloor t/3 \rfloor$ or at most $\lfloor t/3 \rfloor + 1$ processors in such a way that the total number of faulty processors never exceeds t in any of the scenarios. □

Regarding the condition $t \geq 3$ in Theorem 5.2, by using the assumption that the receiver of a message knows the identity of the sender, it is not hard to find a 2-resilient DFS algorithm for any number $n \geq 2$ of processors. If we add the requirement that each correct processor must broadcast one message at each step (formally, if $(m_1, m_2, \ldots, m_n)$ is in the range of β_i then $m_1 = m_2 = \ldots = m_n$), then we can show that there is a t-resilient DFS algorithm iff $n \geq 2t+1$, provided $t \geq 2$. This is because the broadcast condition prevents the correct processors from hiding from the faulty processors text that these faulty processors would otherwise have to forge. (Note that this result applies to communication systems like the Ethernet, in which eavesdropping cannot be avoided.)

6. RELATED RESULTS

6.1. Byzantine Agreement with Nonunison Start

Suppose we want to solve authenticated Byzantine agreement when the correct processors do not all awaken at the same time, but some wake up at time 0 and some wake up at time 1, and the faulty processors can rush. The known efficient algorithm of Dolev and Strong [DS] does not work in this case. One solution would be to first run the DFS algorithm of Theorem 4.1 and then run the Dolev-Strong algorithm, for a total time of 2t+6. This time can be improved to t+5 by modifying the algorithm of Theorem 4.1 to solve agreement directly. However, the algorithm requires $n \geq 3t+1$. We have a completely different solution, not using DFS ideas, which tolerates any number $t \leq n$ of faults but takes 2t+2 rounds. We do not know whether arbitrary fault tolerance and time t+O(1) can be obtained simultaneously.

6.2. Discrete Clock Synchronization

Suppose each processor has a local integer-valued clock which is incremented by 1 at each step. Suppose the clocks of correct processors are initially almost synchronized in that the clocks of two correct processors differ by at most one at any integer global time. We want a protocol which will reset certain clocks so that the clocks of all correct processors read exactly the same at each global time, and the clock of a correct processor is never moved back. The DFS algorithm of Theorem 4.1 can be modified to solve this problem in t+O(1) rounds, provided $n \geq 3t+1$.

6.3. DFS With a Global Clock

Suppose we have a system where the correct processors have access to a common global clock. At each integer time s, all correct processors taking their unison step at time s know that it is time s. In this case, DFS becomes easier but it is not trivial. We can give a reduction similar to that of Theorem 3.2. As corollaries of this reduction, DFS with a global clock still requires t+1 rounds for fail-stop faults, and $n \geq 3t+1$ is needed in the nonauthenticated Byzantine case. In the authenticated Byzantine case, with rushing and collusion, there is a t-resilient algorithm for any $t \leq n$ which halts in t+1 rounds.

REFERENCES

[ADG] Attiya, C., Dolev, D., and Gil, J., Asynchronous Byzantine consensus, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 119-133.

[BD] Broder, A., and Dolev, D., Flipping coins in many pockets (Byzantine agreement on uniformly random values), *Proc. 25th Symp. on Foundations of Computer Science*, 1984, pp. 157-170.

[BL] Burns, J. E., and Lynch, N. A., The Byzantine firing squad problem, manuscript, submitted for publication.

[DDS] Dolev, D., Dwork, C., and Stockmeyer, L., On the minimal synchronism needed for distributed consensus, *Proc. 24th Symp. on Foundations of Computer Science*, 1983, pp. 393-402.

[DFFLS] Dolev, D., Fischer, M.J., Fowler, R., Lynch, N.A., and Strong, H.R., Efficient Byzantine agreement without authentication, *Information and Control* 52 (1982), pp. 257-274.

[DHS] Dolev, D., Halpern, J., and Strong, H. R., On the possibility and impossibility of achieving clock synchronization, *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 504-512.

[DLS] Dwork, C., Lynch, N., and Stockmeyer, L., Consensus in the presence of partial synchrony, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 103-118.

[DRS] Dolev, D., Reischuk, R., and Strong, H. R., Eventual is earlier than immediate, *Proc. 23rd Symp. on Foundations of Computer Science*, 1982, pp. 196-203.

[DS] Dolev, D. and Strong, H. R., Authenticated algorithms for Byzantine agreement, *SIAM J. Computing* 12 (1983), pp. 656-666.

[FLM] Fischer, M. J., Lynch, N. A., and Merritt, M., Shifting scenarios: easy impossibility proofs for distributed consensus problems, manuscript.

[FLP] Fischer, M., Lynch, N. A., and Paterson, M., Impossibility of distributed consensus with one faulty process, *Proc. 2nd Symp. on Principles of Database Systems*, 1983, pp. 1-7.

[HSSD] Halpern J., Simons, B., Strong, H. R., and Dolev, D., Fault-tolerant clock synchronization, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 89-102.

[L] Lamport, L., The weak Byzantine generals problem, *J.ACM* 30 (1983), pp. 668-676.

[LF] Lamport, L., and Fischer, M.J., Byzantine generals and transaction commit protocols, manuscript.

[LL] Lundelius, J., and Lynch, N., A new fault-tolerant algorithm for clock synchronization, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 75-88.

[LM] Lamport, L., and Melliar-Smith, P.M., Byzantine clock synchronization, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 1984, pp. 68-74.

[LSP] Lamport, L., Shostak, R., and Pease, M., The Byzantine generals problem, *ACM Trans. on Programming Languages and Systems* 4 (1982), pp. 382-401.

[Mo] Moore, E. F., The firing squad synchronization problem, in: E. F. Moore, Ed., *Sequential Machines, Selected Papers*, Addison-Wesley, Reading, MA, 1964.

[Me] Merritt, M., personal communication.