SIAM J. COMPUT.
Vol. 18, No. 5, pp. 990-1012, October 1989

010

THE DISTRIBUTED FIRING SQUAD PROBLEM*

BRIAN A. COAN†, DANNY DOLEV‡, CYNTHIA DWORK§¶, AND LARRY STOCKMEYER§

Abstract. The distributed firing squad problem is defined in the context of a synchronous distributed system where the correct processors operate in lock-step synchrony but do not share a global clock. If one or more correct processors receive a command to start a firing squad synchronization, then at some future time all correct processors must "fire" (formally, enter a special state) at exactly the same step. For various fault models, upper and lower bounds are proved on the number of faulty processors that can be tolerated and on the number of rounds of communication required between the reception of the start command and firing. For example, if a firing squad protocol is resilient to t fail-stop faults, then at least $t+1$ rounds are necessary and sufficient. For the case of Byzantine faults with authentication where the faulty processors can take steps in between the synchronous steps of the correct processors, the firing squad problem can be solved in $t+5$ rounds, provided that $n>3t$, where n is the number of processors and t is the number of faults, and the problem cannot be solved at all if $n \leqq 3t$. Moreover, in the case that $n \leqq 3t$, the impossibility of a firing squad protocol holds even for a weaker "timing fault model" where all processors generate messages correctly according to the protocol, but the faulty processors can affect the system by slightly slowing down or speeding up messages.

Key words. firing squad problem, Byzantine generals problem, synchronization, coordination, fault tolerance, distributed computing

AMS(MOS) subject classifications. 68M10, 68M15, 68Q

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., [11], [20], [25]). 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 motivated by the impossibility results of [15] and [8], which show that if the system is asynchronous then there is no protocol for distributed agreement tolerant to even one benign processor failure. There are various ways to maintain synchronous steps in an unreliable distributed system. For example, one can have hardware that sends a periodic signal to all processors. Another common assumption is that all processors begin the algorithm simultaneously, i.e., at the same step. Typically, however, an algorithm is executed in response to a request from some specific processor that 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.

* Received by the editors July 23, 1986; accepted for publication (in revised form) October 17, 1988. A preliminary and abridged version of this paper appeared in the Proceedings of the 17th ACM Symposium on Theory of Computing, Providence, Rhode Island, 1985.

† Bell Communications Research, MRE 2P-252, 445 South St., Morristown, New Jersey 07960. This work was performed in part while the author was at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts. It was supported by the National Science Foundation under grant DCR-8302391, 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.

‡ IBM Research Division, K53/802, 650 Harry Road, San Jose, California 95120, and Computer Science Department, Hebrew University, Jerusalem, Israel. This work was performed in part while the author was a Batsheva de Rothschild Fellow.

§ IBM Research Division, K53/802, 650 Harry Road, San Jose, California 95120.

¶ This work was performed in part while the author was a Bantrell Postdoctoral Fellow at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

In this paper we justify the design assumption of simultaneous starts. Specifically, we provide algorithms to solve the associated synchronization problem that we call the *distributed firing squad* problem. An algorithm for the distributed firing squad problem has two properties:

(1) If any correct processor receives a message to start a firing 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.

Our principal results, which are summarized in this Introduction, are for the case in which the processors, although operating in lock-step, are not assumed to share a common view of the current "global" time. We also present a simpler algorithm for the case in which such a common view is assumed.

The two complexity measures we study are *fault tolerance*, the maximum number of faulty processors that 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 firing 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 sent by correct processors, but only to the extent of distinguishing 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 algorithm correctly but simply stops at some point), it is easy to find a t-resilient distributed firing squad algorithm for any number $t \leqq n$ of faults that halts in $t+1$ rounds. This was observed independently by Burns and Lynch [3]. By reducing the Weak Byzantine Agreement (WBA) problem to the distributed firing squad problem, we can use a lower bound of Lamport and Fischer [18] on the time complexity of the WBA problem to show that any t-resilient algorithm for the distributed firing squad problem requires $t+1$ rounds for fail-stop faults, and therefore also for more malicious types of faults. Thus, the situation for fail-stop faults is well understood. Burns and Lynch [4] give a distributed firing squad 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., [11]), and the signatures are common knowledge.

In trying to determine the maximum fault tolerance of the distributed firing squad problem in the authenticated Byzantine case, we found it necessary to distinguish between several types of faulty behavior, since these distinctions affect the fault tolerance. In *rushing*, a Byzantine faulty processor can receive, process, and resend 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. Fig. 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

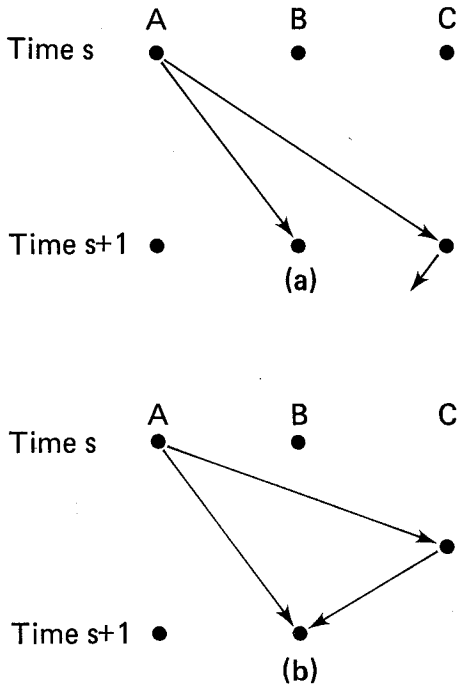

FIG. 1. (a) *A communication round with three correct processors. For simplicity, only the messages sent by A at time s are shown.* (b) *The processor C rushes.*

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 signature functions of other faulty processors, which we call *collusion*, and the case where a faulty processor has only its own signature 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 that allowed faulty processors to share signature keys.

Table 1 summarizes our results and the results of [4] for the different fault models. Each entry gives $n_{\min}$, the smallest number n ($n \geqq 2$) of processors for which there exists a t-resilient distributed firing squad algorithm ($t \geqq 1$). Unless otherwise indicated, all algorithms require at most $t+c$ rounds, where $c \leqq 5$ is a constant independent of n and t, and the total number of bits of communication sent by correct processors 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 because the necessary deductions about the sender's identity can be made in the fault models we consider.

There are several interesting things to note about these results. First we should emphasize that the lower bound $n \geqq 3t+1$ 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. Second, even though our bounds on the

TABLE 1

Fault	$n_{\min}$ for t-resiliency	Remarks
fail-stop	t	1
Byzantine with authentication		
no rushing, no collusion	t	1
collusion, no rushing	$5t/3 < n_{\min} \leqq 2t+1$	2
rushing, no collusion	$3t+1$	
rushing and collusion	$3t+1$	
timing faults	$3t+1$	
Byzantine without authentication	$3t+1$	3

Remarks. (1) *Due independently to Burns and Lynch* [3]. (2) *Lower bound* $5t/3 < n_{\min}$ *proved only for* $t \geqq 3$; *algorithm with* $2t+1$ *processors takes* $2t+1$ *rounds.* (3) *Algorithm due to Burns and Lynch* [4]; *algorithm uses either exponential communication or more than* $t+O(1)$ *rounds. Except as noted in* 2 *and* 3, *running time of all algorithms is* $t+c$, $c \leqq 5$, *and communication complexity is polynomial in n.*

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 distinction thus shown between these three fault models is (to us) an unexpected result of this work.

Burns and Lynch [4] solve the distributed firing squad problem in the unauthenticated Byzantine case essentially by adapting an agreement protocol. Since all known unauthenticated agreement protocols either use exponential communication, use more than $t+O(1)$ rounds, or require $n > 8t$ [2], [5], [9], [10], [20], [24], their distributed firing squad solution has the same property. Recently, Moses and Waarts [24] have devised a new unauthenticated agreement protocol; together with the work of Burns and Lynch, this gives an unauthenticated distributed firing squad protocol using polynomial communication, $n > 8t$ processors, and the optimal time $t+1$. By using signatures, we achieve polynomial communication, time $t+5$, and the maximum resiliency $t = \lfloor (n-1)/3 \rfloor$. Our lower bounds $n > 3t$ ($n > 5t/3$) for rushing (collusion) suggest that the approach of directly adapting agreement protocols to the distributed firing squad problem will not work in the authenticated case, since there are authenticated agreement protocols tolerant to any number of failures [11]. For the same reason, the distributed firing squad problem seems to be different from the clock synchronization problem studied in [16], [19], [21], [26], 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 [16]. Our problem is also different from the version of the firing squad problem that was proposed in the late 1950s [23]. That version of the problem was interesting because the processors were finite state machines which were connected in a linear array so each processor could count only 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 (we assume a completely connected system of powerful processors) but rather from the possibility of processor and timing faults.

In § 2 we give definitions. Section 3 contains results (firing squad algorithm and lower bound) for the case of no collusion and no rushing, § 4 gives results for rushing and timing faults, and § 5 considers the case of collusion but no rushing. In § 6 we mention some related results, such as the application of firing squad ideas to the

problem of Byzantine Agreement in the case that the processors do not all start at the same round, and results concerning the distributed firing squad problem where the processors share a common view of global time.

2. Definitions. For simplicity we give definitions for a single occurrence of a firing squad synchronization. Let $p_1, p_2, \cdots, p_n$ denote the processors in the system. For technical reasons we introduce another "processor" w, whose only purpose is to start a 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 distributed firing squad 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

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

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

$$\sigma_i(q_0, \varnothing, \cdots, \varnothing) = q_0,$$
$$\beta_i(q_0, \varnothing, \cdots, \varnothing) = (\varnothing, \cdots, \varnothing).$$

We introduce the concept of *global time* as an expositional convenience. The individual processors have no knowledge of global time. We assume that processors take steps at global times specified by nonnegative real numbers. A *run* is specified by giving, for each processor $p_1, \cdots, p_n$, w (including both correct and faulty processors), a list of nonnegative real numbers that 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.

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 $(\varnothing, \varnothing, \cdots, \varnothing)$, and thereafter takes steps at successive integer times 1, 2, 3, $\cdots$;

(2) p_i executes its algorithm (transition functions) correctly; and

(3) if authentication is assumed (see below), then no other processor p_j signs any message using the signature function of p_i.

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 integer). 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 $fire_i(R)$ 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).

A distributed firing squad 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 $p_1, \cdots, p_n$ are faulty and in which the faulty processors conform to the given type of faulty behavior, there is a (necessarily integer) time $fire(R) \geqq awake(R)$ such that $fire_i(R) = fire(R)$ for all i such that p_i is correct in R. The *time complexity* of the 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.)

Since the definitions above consider a processor to be faulty throughout a run if it fails at any time in the run, the definitions would seem to allow a scenario S where some processor p_i first fails after time $fire(R)$; thus, the definitions do not require p_i to fire with the other processors even though it is actually "correct" throughout execution of the algorithm. It is sufficient to note, however, that any such scenario can be modified to a scenario S', where p_i is correct throughout the entire (infinite) run, and all processors behave exactly as they do in S from time 0 through time $fire(R)$. In the modified S', p_i violates the definition of correctness. We find it convenient, both for definitions and for proofs of correctness, to consider a processor to be either correct or faulty throughout an entire run rather than to define the first time when a faulty processor actually exhibits faulty behavior.

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 time s, at time s some nonempty 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. A processor is *Byzantine* if its behavior can deviate in any way from the behavior specified by its transition functions. A special case of Byzantine faultiness is Byzantine faults with authentication, where each processor p can sign messages using a private signature function E_p in such a way that the signature of a correct processor cannot be forged by any other processor. In this case, if processor q receives a message $E_p(m)$ from a third processor r, then q knows that if p is correct then p actually sent the message $E_p(m)$ at some previous time. We also let E_i denote the signature function of processor p_i. A Byzantine faulty processor *rushes* if it takes some step at a noninteger time. In this case, messages to and from faulty processors may take less than one round to be delivered. Faulty processors p and q *collude* if p signs a message using the signature function of q. In this case q is considered faulty, even though the messages it sends may be correct.

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 $0, 1, 2, \cdots$;

(3) faulty processors take steps at times $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \cdots$;

(4) messages between two correct processors or between two faulty processors take one unit of time to be delivered; and

(5) messages between a correct and a faulty processor take either $\frac{1}{2}$ unit of time or $\frac{3}{2}$ units of time 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 $\frac{3}{2}$ 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 $\frac{3}{2}$ 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 $\frac{5}{2}$. Therefore, giving an 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.

The *communication complexity* of an algorithm is the maximum total number of message bits sent by correct processors, between the time when a correct processor first awakens and the time when all correct processors fire. The communication complexity is expressed as a function of n. Of course, there is no way to bound the number of bits sent by Byzantine faulty processors. For each of our algorithms it is

easy to see that exceedingly long messages sent by faulty processors never cause a correct processor to send exceedingly long messages. This is true because each correct processor has, at each step of the algorithm, a bound on the length of messages it expects to receive. Once the length of a message exceeds that bound, the message can be ignored.

As the firing squad problem is defined above, any single Byzantine faulty processor can initiate a firing synchronization. One way to limit the ability of faulty processors to start extraneous synchronizations in models with authentication is to have w sign the awake message with its own unforgeable signature. A correct processor considers as null any received message that does not contain w's signature. The necessary changes to our algorithms are trivial (in algorithms that count the number of signatures on a message, the signature of w is not counted). This still allows a faulty w to start extraneous synchronizations, but this is unavoidable in any model where we allow a single processor or an external agent to start a synchronization.

Processors as defined above are deterministic. Coan and Dwork [6] have studied probabilistic protocols for firing squad synchronization and have found that probabilistic protocols are essentially no better than deterministic ones for the firing squad problem.

3. No rushing and no collusion.

3.1. Upper bound. We begin with a simple algorithm that 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 [3]. 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 those of the Dolev-Strong algorithm for authenticated Byzantine agreement [11].

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

Proof. Each processor p participating in the protocol has a private clock c_p that is completely under the control of p. Initially, $c_p = -1$. A message m is *proper* if it has the form

$$m = E_{i_1}(E_{i_2}(\cdots E_{i_k}(\text{Awake})\cdots))$$

where E_{i_j} is the signature function of p_{i_j}, and the k signatures are by distinct processors. The *length* of the proper message m, denoted $|m|$, is the number of signatures appearing in m (i.e., k above). The awake message has length 0. A proper message m is *acceptable* to p if and only if $|m| > c_p$. A proper message m is *new* to p if and only if p's signature does not appear in m.

Upon first awakening, p sets its clock to $|m|$, where m is any acceptable message of maximum length received by p. If $c_p = t+1$, then p fires; otherwise, p signs m and broadcasts the result $E_p(m)$. If all messages received by p in this first step are unacceptable, then p sets its clock to 0 and broadcasts $E_p(\text{Awake})$.

At each subsequent step, if p receives any acceptable message, p arbitrarily chooses one such message m of maximum length and sets c_p to $|m|$. If $c_p \geqq t+1$, then p fires; if $c_p < t+1$ and m is new to p, then p signs m and broadcasts the result $E_p(m)$. If no acceptable message is received, p increments c_p by one. Again, if $c_p \geqq t+1$ then p fires.

This completes the description of the protocol for a correct p.

LEMMA 3.1.1. *For all k with $k \geqq 0$ and for any message m, at least k rounds are required to add k distinct signatures to m.*

Proof. Each processor knows at most one signature function and there is no rushing. □

LEMMA 3.1.2. *In any execution of the protocol resulting in a firing, all correct processors fire simultaneously.*

Proof. Let r be the earliest step at which some correct processor fires, and let p be a correct processor that fires at step r. We will show that for all correct processors q, $c_q \geqq t+1$ by the end of step r, so q fires at r.

There are two cases to consider, according to whether or not p receives an acceptable message at step r. If p receives an acceptable message m at step r, then m has at least $t+1$ signatures. Without loss of generality, let $m = E_k(E_{k-1}(\cdots E_1(\text{Awake})\cdots))$, where $k = |m| \geqq t+1$. Clearly, if p_k is correct then all correct processors receive m at r, and so all correct clocks have value at least $t+1$ by the end of step r. If p_k is faulty, then let p_i be the last correct processor whose signature appears in a substring $m_i = E_i(\cdots E_1(\text{Awake})\cdots)$ of m, and let $s-1$ be the step at which m_i is sent by p_i. By Lemma 3.1.1 at most one signature can be added to m_i in each step, so $r - s \geqq |m| - |m_i|$. Furthermore, since p_i is correct, the clocks of all correct processors have value at least $|m_i|$ by the end of step s. Since the clock of a correct processor is incremented by at least 1 at each step, by the end of step r the clocks of all correct processors will have value at least

$$|m_i| + (r-s) \geqq |m_i| + (|m| - |m_i|) = |m| \geqq t+1$$

so all correct processors will fire at step r.

In the second case where p receives no acceptable message at step r, let i be such that step $r-i$ is the last step at which p broadcasts. (Since this happens when p first awakens, i is well defined.) Let $E_p(m)$ be the message broadcast by p at $r-i$. Since this message has length $|m|+1$ and since all correct processors receive this message at step $r-i+1$, by the end of step $r-i+1$ all correct processors have clock value at least $|m|+1$. Therefore, the clocks of all correct processors have value at least $|m|+i$ by the end of step r. We now want to argue that p increments its clock by exactly 1 at each step j with $r-i+1 \leqq j \leqq r$. Suppose otherwise that p increments its clock by more than 1 at step j, and let m' be the acceptable message received at step j which causes this increment. Since p receives no acceptable message at step r, we must have $j < r$. If m' is not new to p at step j, then using Lemma 3.1.1 as in the preceding paragraph, it is easy to show that m' could not cause p to increment its clock by more than 1. If m' is new to p, then by definition of the algorithm p would broadcast at step j, contradicting the choice of i. Therefore, such a j and m' cannot exist. Having bounded p's clock increment at each step j with $r-i+1 \leqq j \leqq r$, it follows that $c_p = |m|+i$ at the end of step r. Since p fires at r, we have $|m|+i \geqq t+1$. Thus all correct processors fire at step r. □

LEMMA 3.1.3. *Let R be any active run and let s be the earliest time when some correct processor p awakens in R. Then p fires at time $s+t+1$ or earlier.*

Proof. Upon awakening, p sets its clock to some value $c_p \geqq 0$. At every step c_p is incremented by at least one, and p fires when $c_p \geqq t+1$. □

It is now easy to complete the proof of Theorem 3.1. By Lemma 3.1.3, if any correct processor awakens there is a firing, and by Lemma 3.1.2 all correct processors fire simultaneously. Lemma 3.1.3 also implies that the time complexity of the algorithm

is at most $t+1$ rounds. Furthermore, at each step each correct processor broadcasts at most one message. Since this message is proper, it requires only polynomial in n bits. Thus the communication required is polynomial in n. □

Obviously, the algorithm of Theorem 3.1 also works for fail-stop faults: instead of signing a message m with the signature function E_p, the processor p simply attaches its name to m.

3.2. Lower bound. We now show that the time complexity of this algorithm is optimal by reducing the Weak Byzantine Agreement problem (WBA) [17] to the distributed firing squad problem. Optimality follows from the fact that WBA requires at least $t+1$ rounds [18].

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. 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 shows that WBA reduces to distributed firing squad at no cost in running time.

THEOREM 3.2. *Let A be a distributed firing squad algorithm that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that 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 *that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that 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, p_i 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 $\leqq k$ 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) *Let $t \leqq n-2$. Any distributed firing squad algorithm 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 round is fixed a priori. It is also true even in all executions in which all processors are correct.*

(2) *Any distributed firing squad algorithm resilient to t unauthenticated Byzantine faults requires at least $3t+1$ processors.*

Proof. The proof is immediate from the preceding theorem and the corresponding bounds for WBA [13], [14], [17], [18], [22]. □

Remark. To close the gap between Theorem 3.1 and Corollary 3.3(1) for $n-1 \leqq t \leqq n$, note that if we take $t = n-2$ in the algorithm of Theorem 3.1, then the algorithm is in fact n-resilient. If there is only one correct processor p, then p will fire within $t+1$

$(=n-1)$ steps after awakening, since c_p is incremented by at least 1 at each step. If there are no correct processors, then there is nothing to prove.

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

4. Rushing and timing faults.

4.1. Upper bound. This section contains tight bounds on fault tolerance for timing faults and authenticated Byzantine faults 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 algorithm requiring* $t+5$ *rounds,* $n \geqq 3t+1$ *processors, and communication polynomial in n.*

Before giving the details of the algorithm and its proof of correctness, we begin with an informal discussion of the principal ideas. Our protocol is composed of a set of identical 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 [11] that 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 orders a firing only if there are at least $n-t \geqq 2t+1$ processors in the agreed upon set. Of these, at least $t+1$ are correct, synchronized processors. Thus a correct processor orders a firing only 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. An execution of the 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. An execution of the 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 that 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) that receives $E_p(p)$ signs it and broadcasts it. Each processor q then attempts to form a *core for p,* that is, a list of the form

$$\langle E_{i_1}(E_p(p)), \cdots, E_{i_k}(E_p(p))\rangle$$

where $k \geq n-t$ and each of the k copies of $E_p(p)$ is signed by a distinct processor. The signatures $E_{i_1}, \cdots, E_{i_k}$ *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 time.

A *notarized core for p* is a list of the form

$$\langle E_{i_1}(C_1), \cdots, E_{i_k}(C_k)\rangle$$

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

We now describe how a processor q participates in the core protocol initiated by p. (Processor p participates in the core protocol initiated by itself.) Let s be the global time when q receives $E_p(p)$. Then q tries to form a core for p at time $s+1$. This is done by looking for a set of messages $\{E_{i_1}(E_p(p)), \cdots, E_{i_k}(E_p(p))\}$ received at time $s+1$ where $k \geq n-t$ and where each of the k copies of $E_p(p)$ is signed by a distinct processor. This is the only time at which q tries to form a core for p, and q includes in the core only 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 $s+2$. This is the only time when q tries to form a notarized core for p. 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 correct processor q does the following (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) $m = E_{j_1}(E_{j_2}(\cdots E_{j_k}(N)\cdots))$, where N is a notarized core for p and each of the k signatures ($k \geq 0$) is 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 back (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 $E_{j_1}, E_{j_2}, \cdots, E_{j_k}$ is by q. If q finds one or more messages of length k that 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 $r+2$. Furthermore, 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 for p 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, $0 \leqq i \leqq t$, 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 is true inductively for $i-1$ ($i \geqq 1$). 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 $n-t \geqq 2t+1$ 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 of length $t+1$, so all correct processors broadcast "fire$_p$". 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 at which the members of G receive $E_p(p)$. 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 $|H| \leqq t$. 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). Furthermore, 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 that 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

$$E_q(C) = E_q(\langle \cdots, E_g(E_p(p)), \cdots \rangle).$$

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 that 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

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

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

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

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 $z+1$. Furthermore, by Lemma 4.1.8, G contains at least $n-2t \geq t+1$ processors, so there will be

some processor in G that has not yet signed N, provided $|E_c(m')| \leqq t$. 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. □

4.2. Lower bound. We next give a matching lower bound, $n \geqq 3t+1$, for the timing fault model. As noted in § 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 distributed firing squad algorithm only if* $n \geqq 3t+1$.

Proof. Consider first the proof that there is no 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 Fig. 2) so that the following lemmas hold.

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. □

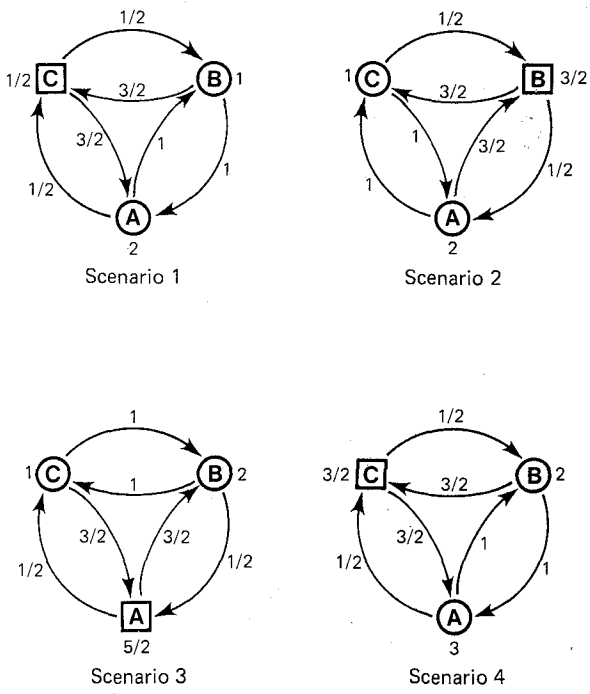

FIG. 2. *The scenarios used to prove Theorem* 4.2. *The number on the edge directed from processor X to processor Y is the message transmission time from X to Y. The number written next to processor X is the time when processor X wakes up. Correct (faulty) processors are drawn as circles (squares).*

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 Fig. 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 Fig. 2, one can easily verify that the following two facts hold for all four scenarios and for all integers $s \geqq 1$:

(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 $n \leqq 3t$ is done as usual (cf. [25]) 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 Fig. 2. This completes the proof of Theorem 4.2. □

5. Collusion.

5.1. Upper bound. 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 distributed firing squad algorithm requiring $n \geqq 2t+1$ processors, $2t+1$ rounds, and an amount of communication polynomial in n.*

We describe the main ideas informally before giving the formal proof. As in the other protocols, 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 $t+1$ acknowledgments of the form $E_q(E_p(m))$. The list

$$m' = \langle E_{q_1}(E_p(m)), \cdots, E_{q_{t+1}}(E_p(m)) \rangle$$

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, since each notarization increases the size of a message by at least the factor $t+1$. Our algorithm uses the idea of notarization with an implementation that is harder to prove correct but that requires communication only polynomial in n. The basic idea is as follows. Suppose that p has received a notarized message m in the form of at least $t+1$ signatures of m by distinct processors. Then p "requests support" of $E_p(m)$ by broadcasting $E_p(m)$ together with a "proof" that m was notarized. This proof consists of $t+1$ signatures of m by distinct processors. Any correct processor q receiving $E_p(m)$, together with such a proof, "supports" $E_p(m)$ by signing $E_p(m)$ and broadcasting the result. The key fact is that the proof that m was notarized can be thrown away by q at this point, so message length does not grow exponentially. The idea of notarization and its implementation below is similar to the fault-tolerant distributed clocks described in [1], [12]. Similar ideas were also used in [27].

Proof of Theorem 5.1. We first define certain types of messages. A *proper* message has the form

$$E_{i_1}(E_{i_2}(\cdots E_{i_k}(p_{i_k})\cdots))$$

where the k signatures are by distinct processors. Such a message is called an *α_k-message.*

At various times in the protocol, processors may request support for a message m. A processor p *supports* m by sending a *support message* of the form $E_p(\text{support } m)$. We let $S(m)$ denote a support message for m.

A *proof* of a message m is a list of $t+1$ support messages for m, each signed by a distinct processor. We let $P(m)$ denote a proof for message m.

A processor p *requests support* for a message $E_p(m)$ by broadcasting a message of the form $\langle E_p(m), P(m)\rangle$. We let $R(E_p(m))$ denote a request for support of $E_p(m)$. When $E_p(m)$ has the form α_k, $k>1$, we call this request an *R_k-message.* An R_1-message has the form $\langle E_p(p), \lambda\rangle$, where λ denotes the empty string.

We now describe the protocol for a correct processor p. At every step p may issue both support messages and requests for support. In particular, after receiving at each step, p does the following.

(1) p chooses the maximum i such that p can form an R_i-message, $R(E_p(m)) = \langle E_p(m), P(m)\rangle$, where $E_p(m)$ is proper and p could not form an R_i-message at any previous step. If such an i exists then p broadcasts an R_i-message. If it can construct several syntactically distinct R_i-messages, then it arbitrarily chooses one to broadcast.

(2) For each processor q, p chooses the maximum j such that p receives an R_j-message $\langle E_q(m), P(m)\rangle$ from q. If $j<t+1$, then p broadcasts the corresponding support message $E_p(\text{support } E_q(m))$.

(3) If p receives an R_j-message for some $j\geqq t+1$, then p fires.

Viewing the number of signatures on a message as a clock, the two key lemmas state that the faulty processors cannot increment the clock faster than by 1 within two steps (Lemma 5.1.1) and that the correct processors can increment the clock at least that quickly (Lemma 5.1.2).

LEMMA 5.1.1. *Let $R(m)$ be an R_i-message and let s be the earliest time at which some correct processor sends $R(m)$. Let s' be the earliest time at which some (possibly faulty) processor q sends $R(m')$, where m' is an α_j-message of the form $E_b(E_c(\cdots(m))$. Then $s'\geqq s+2(j-i)$.*

Proof. The proof is by induction on $j-i$.

Basis $j-i=0$. Since $R(m)$ is sent by a correct processor, m is of the form $E_p(m'')$ for some correct processor p. Since $m'=m$ in this case, and since q cannot forge p's signature, q cannot construct $R(m')$ before time s.

Assume the lemma inductively for $j-i \leqq k$. Let $j=i+k+1$. Let $m''=E_c(\cdots(m))$ be such that $E_b(m'')=m'$. By the inductive hypothesis $R(m'')$ can be constructed no sooner than step $s+2(j-i-1)$.

If q constructs $R(m')$ at s', then q constructs $P(m'')$ at s', so q receives $t+1$ $S(m'')$ messages no later than time s'. Thus some correct processor received $R(m'')$ at $s'-1$, so $R(m'')$ was sent not later than $s'-2$. We therefore have $s'-2 \geqq s+2(j-i-1)$ by the inductive hypothesis, so $s' \geqq s+2(j-i)$ and the induction holds. □

LEMMA 5.1.2. *Fix a time s. Let i be the maximum i such that a correct processor broadcasts an R_i-message at time s, and let p be such a processor. If $i<t+1$, then by time $s+2$ some correct processor q forms and broadcasts an R_j-message for some $j \geqq i+1$.*

Proof. Without loss of generality, we may assume s is the first time at which p can construct an R_i-message. In this case p broadcasts $R(E_p(m))=\langle E_p(m), P(m)\rangle$ at s ($\langle E_p(m), \lambda\rangle$ if $i=1$), where $E_p(m)$ is an α_i-message, and every processor receives $R(E_p(m))$ at time $s+1$. Since this is the only request for support sent by p at s, every correct processor responds by broadcasting $S(E_p(m))$ at $s+1$. Thus all processors receive $n-t \geqq t+1$ $S(E_p(m))$ messages at $s+2$, each signed by a distinct processor, so at time $s+2$ every correct processor can construct a proof for $E_p(m)$. By signing $E_p(m)$ to obtain an α_{i+1}-message and combining this with the proof for $E_p(m)$, any processor that has not already signed $E_p(m)$ can construct an R_{i+1}-message. Because $i<t+1$, there is at least one such correct processor, say, q. If q has already broadcast an R_j-message for some $j \geqq i+1$, then the lemma holds trivially. Otherwise, by Step 1 of the protocol and the fact that it can construct an R_{i+1}-message, q will broadcast an R_j-message for some $j \geqq i+1$ at time $s+2$. □

LEMMA 5.1.3. *If any correct processor awakens, then every correct processor eventually fires. Moreover, if s is the earliest time when some correct processor awakens, then every correct processor fires by time $s+2t+1$.*

Proof. Upon awakening, each correct processor forms and broadcasts an R_j-message, for some $j \geqq 1$, since the proof of an R_1-message is the empty string. Let p be the first correct processor to awaken, and let s be the time at which it awakens. By t applications of Lemma 5.1.2, some correct processor constructs and broadcasts an R_j-message for some $j \geqq t+1$ by time $s+2t$. Thus every correct processor fires by $s+2t+1$. □

LEMMA 5.1.4. *In any execution of the protocol resulting in a firing, all correct processors fire simultaneously.*

Proof. Let p be the first correct processor to fire and let f be the time at which p fires. If p fires at f then, for some $k \geqq t+1$, p receives an R_k-message at f. Let $R(m')$ be such a message. Without loss of generality let $m'=E_k(\cdots E_2(E_1(p_1)))$. Let i be the maximum i, $1 \leqq i \leqq k$, such that p_i is correct. (Since there are at most t faulty processors, some such p_i exists.) If $p_i=p_k$, then all processors receive $R(m')$ at f so all fire simultaneously at f.

If $i<k$, then at some round $f'<f$, p_i broadcast an R_i-message. This message was received by all correct processors no later than round $f'+1 \leqq f$. If $i \geqq t+1$, then by Step 3 of the protocol all correct processors fire at $f'+1$. Since f is the first round at which any correct processor fires, we have $f'+1=f$, and all correct processors fire simultaneously.

We now consider the case $i<t+1$. Let s be the time at which p_i broadcasts $R(m)$, where $m=E_i(\cdots(E_1(p_1)))$. Since p_i is correct, s is the earliest time at which $R(m)$ is

sent. By Lemma 5.1.1, no processor can construct $R(m')$ before step $s+2(t+1-i)$, so $f \geqq s+2(t+1-i)+1$. By $t+1-i$ applications of Lemma 5.1.2, some correct processor constructs and broadcasts an R_j-message for some $j \geqq t+1$ by time $s+2(t+1-i)$, so every correct processor receives such an R_j-message by time $s+2(t+1-i)+1$. Thus all correct processors fire by time $s+2(t+1-i)+1$. Since p is the first correct processor to fire and p fires at f, we have $f \leqq s+2(t+1-i)+1$. Thus all correct processors fire simultaneously at f. □

It is now easy to complete the proof of Theorem 5.1. By Lemmas 5.1.3 and 5.1.4, the algorithm is t-resilient. By Lemma 5.1.3, every correct processor fires within $2t+1$ rounds after the first correct processor awakens. Finally, at each step a correct processor broadcasts at most one request for support and n support messages, so at each step the number of bits sent by any correct processor is polynomial in n. Since there are n processors and $O(n)$ steps, the total amount of communication is polynomial in n. □

5.2. Lower bound.

THEOREM 5.2. *In the fault model of Theorem* 5.1 *(Byzantine faults with authentication, collusion, but no rushing), if* $t \geqq 3$, *there is a* t*-resilient distributed firing squad algorithm only if* $n \geqq \lfloor 5t/3 \rfloor + 1$.

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

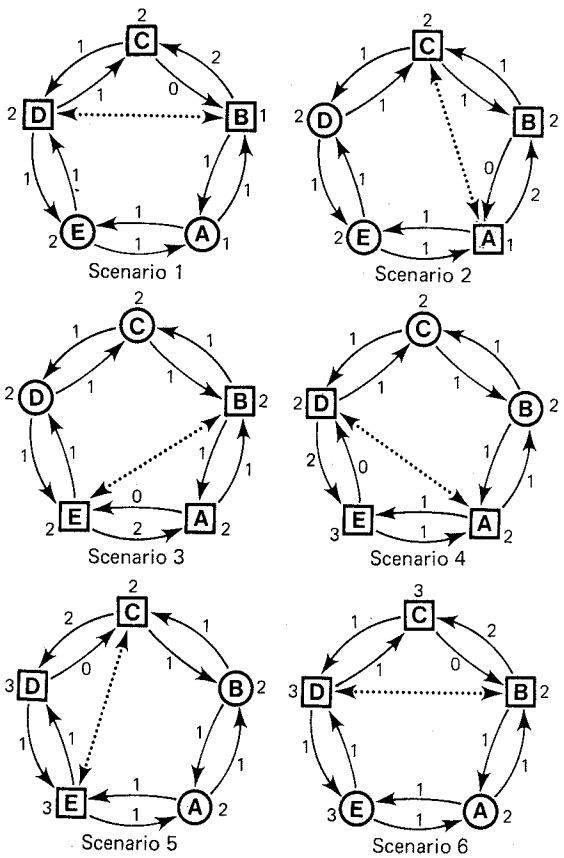

FIG. 3. *The scenarios used to prove Theorem* 5.2.

times, wake-up times, and which processors are faulty in each scenario. A link that 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 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. □

Remark. Regarding the condition $t \geqq 3$ 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 distributed firing squad algorithm for any number $n \geqq 2$ of processors. Briefly, the algorithm is a modification of the algorithm used to prove Theorem 3.1 in the case of no collusion and no rushing. In the modified algorithm, whenever a processor p sends a message m to a processor q, p attaches a header to m that says "the next signer of this message should be processor q". When checking acceptability of a proper message

$$m = E_{i_1}(E_{i_2}(\cdots E_{i_k}(\text{Awake})\cdots))$$

p also checks that m was received from processor p_{i_1} and that each signature in m matches the header of the message being signed. This effectively prevents two faulty processors from adding two signatures in one step, even if they collude. Given this observation, the correctness proof is identical to that of Theorem 3.1, and details are left to the reader. (Note, however, that the modification does not prevent three faulty processors from adding three signatures in two steps, so this method does not generalize to $t \geqq 3$.)

Remark. We can close the fault-tolerance gap between Theorems 5.1 and 5.2 by adding the requirement that each correct processor must broadcast one message at each step (formally, if $(m_1, m_2, \cdots, m_n)$ is in the range of some sending function β_i, then $m_1 = m_2 = \cdots = m_n$). Note that the algorithm of Theorem 5.1 meets this requirement. With this requirement, the proof of the lower bound, Theorem 5.2, can be done with a ring of four processors, two of which are faulty, thus improving the lower bound to $n \geqq 2t+1$. This can be done because the broadcast condition prevents the correct processors from hiding from the faulty processors signed text that these faulty processors would otherwise have to forge. Details are left to the reader. (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 and the faulty processors can rush. For simplicity, we consider the version of the Byzantine agreement problem as in [11], where the protocol is initiated by a single "sender" processor p that wants to send a "value" v to all the processors. If the sender p is correct, then p sends $E_p(v)$ to all processors at exactly the same step; in this case, we require that all correct processors eventually decide that v was the value sent by p. If the sender p is faulty, it can initially send different values to different processors, and it can send values at different times; in this case, we require that if any correct processor decides on a value u, then all correct processors must decide on u. The second type of faulty behavior, initiating the protocol at different times with respect to different processors, is not considered in Dolev and Strong [11], and their efficient ($t+1$ round) algorithm does not work in this case. An obvious solution would be to first run the firing squad algorithm of Theorem 4.1 to synchronize the processors 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.

THEOREM 6.1. *In the model with Byzantine failures and authentication, in which faulty processors can both rush and collude, there is a t-resilient protocol for Byzantine agreement with nonunison start, requiring $t+5$ rounds, $n \geqq 3t+1$ processors, and communication polynomial in n.*

Proof. The algorithm of Theorem 4.1 is modified by associating a value with every core and with every notarized core. Let h be a function that maps a set of values to a single value as follows: $h(\{v\}) = v$; if S is not a singleton set, then $h(S) = 0$. The value associated with the core

$$\langle E_{i_1}(E_p(v_1)), \cdots, E_{i_k}(E_p(v_k))\rangle$$

is $h(\{v_1, \cdots, v_k\})$. The value of the notarized core

$$\langle E_{i_1}(C_1), \cdots, E_{i_k}(C_k)\rangle$$

is h applied to the set containing the values of the cores $C_1, \cdots, C_k$. Each processor q remembers the set V_q of values of notarized cores that it has seen in acceptable messages. In addition to the previous algorithm for signing and forwarding acceptable messages, whenever q receives an acceptable message m containing the notarized core N, and if the value of N is not currently in V_q, then that value is added to V_q and q signs m and broadcasts the result $E_q(m)$. At the point where q receives an acceptable message of length $t+1$, q signs and broadcasts "decide$_p$ $h(V_q)$". A processor decides v if it receives, at the same step, at least $t+1$ messages "decide$_p$ v" signed by different processors. The correctness proof is very similar to the correctness proof given in § 4; only Lemma 4.1.10 requires modification. Details are left to the reader. □

A similar modification to the algorithm solves the version of the agreement problem where each processor begins the algorithm with an initial value.

A drawback in modifying a distributed firing squad algorithm to solve agreement is that it requires $n > 3t$. By adapting an algorithm of Cristian, Aghili, Strong, and Dolev [7] to the model used in this paper, there is a completely different solution, not using firing squad ideas, which tolerates any number $t \leqq n$ of faults but which takes

$2t+2$ rounds. It is an open question whether arbitrary fault tolerance and time $t+O(1)$ can be obtained simultaneously.

6.2. Distributed firing squad with a global clock. At this point, one might suspect that the difficulty of the distributed firing squad problem is due to the processors having no common notion of global time. We show now, however, that the problem does not become trivial, even with a common clock, although for one fault model the problem does become easier. Informally, a common clock means that at each integer time s, all correct processors taking their unison step at time s know that it is time s. (More formally, the state set Q_i of p_i is partitioned into sets $Q_{i,j}$ for integer $j \geqq 0$, and all transitions from a state in $Q_{i,j}$ must go to states in $Q_{i,j+1}$. Each set $Q_{i,j}$ contains a copy $q_{0,j}$ of the quiescent state. The initial state of p_i is $q_{0,0}$. If p_i is in state $q_{0,j}$ and receives only null messages, then p_i next enters state $q_{0,j+1}$.)

Let the *clocked distributed firing squad* problem be defined like the distributed firing squad problem, but in the model with a common clock. We first give a reduction similar to that of Theorem 3.2. As corollaries of this reduction, clocked distributed firing squad still requires $t+1$ rounds for fail-stop faults, and $n \geqq 3t+1$ is needed in the unauthenticated Byzantine case.

THEOREM 6.2. *Let A be an algorithm for clocked distributed firing squad that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that has time complexity of k rounds. Then there exists an algorithm for* WBA *that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that always halts in k rounds.*

Proof. Given A, define the function f on the natural numbers as follows. For a given integer $r \geqq 0$, consider the run of A in which all processors wake up at (common) time r and there are no faults; then $f(r)$ is the (common) time when all processors fire. Since $f(r) \geqq r$ for all r, there must be a time s such that $f(s+1) > f(s)$.

Now 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 as though it were time s. That is, at time 0 of the WBA algorithm, p_i acts as though it were in state $q_{0,s}$ receiving the awake message from w and null messages from the rest. If $v_i = 1$, then p_i waits one step during the WBA algorithm and begins simulating A as though it were awakened at time $s+1$. (In general, time i in the WBA algorithm corresponds to time $s+i$ in the simulation of A.) Let $m = f(s) - s$. If the simulation of A causes p_i to fire within m steps after the beginning of the WBA algorithm, then p_i decides 0 at time m; otherwise, p_i decides 1 at time m. Since A has time complexity k, we have $m \leqq k$. The correctness proof for this WBA algorithm is very similar to the proof of Theorem 3.2 and is left to the reader. □

The next result concerns the case of Byzantine faults with authentication and rushing and shows that the clocked version of the problem is easier for this fault model; specifically, the fault-tolerance improves to any $t \leqq n$, and the time is optimal.

THEOREM 6.3. *In the model with Byzantine failures and authentication, in which faulty processors can both rush and collude, there is a t-resilient clocked distributed firing squad protocol requiring $t+1$ rounds, $n \geqq t$ processors, and communication polynomial in n.*

Proof. In this algorithm, a *proper* message has the form

$$m = E_{i_1}(E_{i_2}(\cdots E_{i_k}(\text{“fire at } c\text{”})\cdots))$$

where c is a natural number modulo $t+1$, where $k \geqq 1$, and where the k signatures are by distinct processors; the *length* of m is k and the *content* of m is c. Such a

message is *acceptable* at common time r if and only if $r \equiv c + k \bmod (t+1)$. This message is *new* to p if p's signature does not appear in m.

A processor p that is awakened by the Awake message at common time s computes $c = s \bmod (t+1)$ and broadcasts E_p("fire at c"). If any processor q receives one or more new acceptable messages with content c at some time, q arbitrarily chooses one, say m, and broadcasts $E_q(m)$. A processor q fires at common time z if q has received, at time z or earlier, an acceptable message m with content c, where $c \equiv z \bmod (t+1)$ such that message m has not caused q to fire at any time earlier than z.

It is clear that if some correct processor awakens at common time s, then all correct processors fire on or before common time $s+t+1$. To argue that all correct processors fire together, we note that if m is new and acceptable to some correct p at time r, then $E_p(m)$ is acceptable to all correct processors at time $r+1$. Let z be the earliest time when some correct processor fires, and let p be a correct processor that fires at z. Therefore, p received an acceptable message m with content c, where $c \equiv z \bmod (t+1)$. If m was received before time z, then all correct processors receive an acceptable message with content c on or before time z, because p must have broadcast such a message before time z. If m is received by p at time z, then $z \equiv c + k \bmod (t+1)$ where k is the length of m, because m is acceptable at time z. Since $c \equiv z \bmod (t+1)$ and $k \geq 1$, it follows that $k \geq t+1$. So m must contain the signatures of $t+1$ processors, at least one of which is correct, and again it is easy to argue that all correct processors received an acceptable message with content c by common time z. □

Acknowledgment. We are grateful to Nancy Lynch for saving an extra round in our reduction of the WBA problem mentioned in § 3.2.

REFERENCES

[1] C. ATTIYA, D. DOLEV, AND J. GIL, *Asynchronous Byzantine consensus*, Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 119-133.

[2] A. BAR-NOY AND D. DOLEV, *Families of consensus algorithms*, Proc. Aegean Workshop on Computing, Greece, 1988, pp. 380-390.

[3] J. E. BURNS AND N. A. LYNCH, personal communication, 1984.

[4] ———, *The Byzantine firing squad problem*, in Advances in Computing Research: Parallel and Distributed Computing, Vol. 4, JAI Press Inc., Greenwich, CT, 1987, pp. 147-161.

[5] B. A. COAN, *A communication-efficient canonical form for fault-tolerant distributed protocols*, Proc. 5th ACM Symposium on Principles of Distributed Computing, 1986, pp. 63-72.

[6] B. A. COAN AND C. DWORK, *Simultaneity is harder than agreement*, Proc. 5th IEEE Symposium on Reliability in Distributed Software and Database Systems, 1986, pp. 141-150.

[7] F. CRISTIAN, H. AGHILI, R. STRONG, AND D. DOLEV, *Atomic broadcast: From simple message diffusion to Byzantine agreement*, Proc. 15th International Conference on Fault Tolerant Computing, 1985, pp. 1-7.

[8] D. DOLEV, C. DWORK, AND L. STOCKMEYER, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77-97.

[9] D. DOLEV, M. J. FISCHER, R. FOWLER, N. A. LYNCH, AND H. R. STRONG, *Efficient Byzantine agreement without authentication*, Inform. and Control, 52 (1982), pp. 257-274.

[10] D. DOLEV, R. REISCHUK, AND H. R. STRONG, *Eventual is earlier than immediate*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 196-203.

[11] D. DOLEV AND H. R. STRONG, *Authenticated algorithms for Byzantine agreement*, SIAM J. Comput., 12 (1983), pp. 656-666.

[12] C. DWORK, N. LYNCH, AND L. STOCKMEYER, *Consensus in the presence of partial synchrony*, J. Assoc. Comput. Mach., 35 (1988), pp. 288-323.

[13] C. DWORK AND Y. MOSES, *Knowledge and common knowledge in Byzantine environments* I: *Crash failures*, Proc. Conference on Theoretical Aspects of Reasoning About Knowledge, Morgan-Kaufmann, Los Altos, CA, 1986, pp. 149-170.

[14] M. J. FISCHER, N. A. LYNCH, AND M. MERRITT, *Easy impossibility proofs for distributed consensus problems*, Distributed Computing, 1 (1986), pp. 26–39.

[15] M. J. FISCHER, N. A. LYNCH, AND M. PATERSON, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374–382.

[16] J. HALPERN, B. SIMONS, H. R. STRONG, AND D. DOLEV, *Fault-tolerant clock synchronization*, Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 89–102.

[17] L. LAMPORT, *The weak Byzantine generals problem*, J. Assoc. Comput. Mach., 30 (1983), pp. 668–676.

[18] L. LAMPORT AND M. J. FISCHER, *Byzantine generals and transaction commit protocols*, Tech. Report Op. 62, SRI International, Menlo Park, CA, 1982.

[19] L. LAMPORT AND P. M. MELLIAR-SMITH, *Synchronizing clocks in the presence of faults*, J. Assoc. Comput. Mach., 32 (1985), pp. 52–78.

[20] L. LAMPORT, R. SHOSTAK, AND M. PEASE, *The Byzantine generals problem*, ACM Trans. Programming Languages and Systems, 4 (1982), pp. 382–401.

[21] J. LUNDELIUS WELCH AND N. LYNCH, *A new fault-tolerant algorithm for clock synchronization*, Inform. and Comput., 77 (1988), pp. 1–36.

[22] M. MERRITT, personal communication, 1984.

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

[24] Y. MOSES AND O. WAARTS, *Coordinated traversal: $(t+1)$-round Byzantine agreement in polynomial time*, Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 246–255.

[25] M. PEASE, R. SHOSTAK, AND L. LAMPORT, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228–234.

[26] T. K. SRIKANTH AND S. TOUEG, *Optimal clock synchronization*, J. Assoc. Comput. Mach., 34 (1987), pp. 626–645.

[27] ———, *Byzantine agreement made simple: Simulating authentication without signatures*, Distributed Computing, 2 (1987), pp. 80–94.