Consensus in the Presence of Partial Synchrony

CYNTHIA DWORK AND NANCY LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

LARRY STOCKMEYER

IBM Almaden Research Center, San Jose, California

Abstract. The concept of partial synchrony in a distributed system is introduced. Partial synchrony lies between the cases of a synchronous system and an asynchronous system. In a synchronous system, there is a known fixed upper bound Δ on the time required for a message to be sent from one processor to another and a known fixed upper bound Φ on the relative speeds of different processors. In an asynchronous system no fixed upper bounds Δ and Φ exist. In one version of partial synchrony, fixed bounds Δ and Φ exist, but they are not known a priori. The problem is to design protocols that work correctly in the partially synchronous system regardless of the actual values of the bounds Δ and Φ. In another version of partial synchrony, the bounds are known, but are only guaranteed to hold starting at some unknown time T, and protocols must be designed to work correctly regardless of when time T occurs. Fault-tolerant consensus protocols are given for various cases of partial synchrony and various fault models. Lower bounds that show in most cases that our protocols are optimal with respect to the number of faults tolerated are also given. Our consensus protocols for partially synchronous processors use new protocols for fault-tolerant "distributed clocks" that allow partially synchronous processors to reach some approximately common notion of time.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; *network operating systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and serviceability*; H.2.4 [**Database Management**]: Systems—*distributed systems*

General Terms: Algorithms, Performance, Reliability, Theory, Verification

Additional Key Words and Phrases: Agreement problem, Byzantine Generals problem, commit problem, consensus problem, distributed clock, distributed computing, fault tolerance, partially synchronous system

---

A preliminary version of this paper appears in *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 103–118.

The work of C. Dwork was supported by a Bantrell postdoctoral Fellowship. The work of N. Lynch was supported in part by the Defense Advance Research Projects Agency under contract N00014-83-K-0125, the National Science Foundation under grants DCR 83-02391 and MCS 83-06854, the Office of Army Research under Contract DAAG29-84-K-0058, and the Office of Naval Research under contract N00014-85-K-0168.

Authors' addresses: C. Dwork and L. Stockmeyer, Department K53/802, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120; N. Lynch, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

1. *Introduction*

1.1 BACKGROUND. The role of synchronism in distributed computing has recently received considerable attention [1, 4, 10]. One method of comparing two models with differing amounts or types of synchronism is to examine a specific problem in both models. Because of its fundamental role in distributed computing, the problem chosen is often that of reaching agreement. (See [8] for a survey; see also [6], [11], [12], and [18] for example.) One version of this problem considers a collection of N processors, $p_1, \ldots, p_N$, which communicate by sending messages to one another. Initially each processor p_i has a value v_i drawn from some domain V of values, and the correct processors must all decide on the same value; moreover, if the initial values are all the same, say v, then v must be the common decision. In addition, the consensus protocol should operate correctly if some of the processors are faulty, for example, if they crash (fail-stop faults), fail to send or receive messages when they should (omission faults), or send erroneous messages (Byzantine faults).

Fix a particular type of fault. Given assumptions about the synchronism of the message system and the processors, one can characterize the model by its *resiliency*, the maximum number of faults that can be tolerated in any protocol in the given model. For example, it might be assumed that there is a fixed upper bound Δ on the time for messages to be delivered (*communication is synchronous*) and a fixed upper bound Φ on the rate at which one processor's clock can run faster than another's (*processors are synchronous*), and that these bounds are known a priori and can be "built into" the protocol. In this case N-resilient consensus protocols exist for Byzantine failures with authentication [3, 15] and, therefore, also for fail-stop and omission failures; in other words, any number of faults can be tolerated. For Byzantine faults without authentication, t-resilient consensus is possible iff $N \geq 3t + 1$ [14, 15].

Recent work has shown that the existence of both bounds Δ and Φ is necessary to achieve any resiliency, even under the weakest type of faults. Dolev et al. [4], building on earlier work of Fischer et al. [10], prove that if either a fixed upper bound Δ on message delivery time does not exist (*communication is asynchronous*) or a fixed upper bound Φ on relative processor speeds does not exist (*processors are asynchronous*), then there is no consensus protocol resilient to even one fail-stop fault.

In this paper we define and study practically motivated models that lie between the completely synchronous and completely asynchronous cases.

1.2 PARTIALLY SYNCHRONOUS COMMUNICATION. We first consider the case in which processors are completely synchronous (i.e., $\Phi = 1$) and communication lies "between" synchronous and asynchronous. There are at least two natural ways in which communication might be partially synchronous.

One reasonable situation could be that an upper bound Δ on message delivery time exists, but we do not know what it is a priori. On the one hand, the impossibility results of [4] and [10] do not apply since communication is, in fact, synchronous. On the other hand, participating processors in the known consensus protocols need to know Δ in order to know how long to wait during each round of message exchange. Of course, it is possible to pick some arbitrary Δ to use in designing the protocol, and say that, whenever a message takes longer than this Δ, then either the sender or the receiver is considered to be faulty. This is not an acceptable solution to the problem since, if we picked Δ too small, all the processors could soon be considered faulty, and by definition the decisions of faulty processors do not have to be consistent with the decision of any other processor. What we would like is a protocol that does not have Δ "built in." Such a protocol would operate correctly whenever it is executed in a system where some fixed upper bound Δ exists. It should also be mentioned that we do not assume any probability distribution on message transmission time that would allow Δ to be estimated by doing experiments.

Another situation could be that we know Δ, but the message system is sometimes unreliable, delivering messages late or not at all. As noted above, we do not want to consider a late or lost message as a processor fault. However, without any further constraint on the message system, this "unreliable" message system is at least as bad as a completely asynchronous one, and the impossibility results of [4] apply. Therefore, we impose an additional constraint: For each execution there is a *global stabilization time* (*GST*), unknown to the processors, such that the message system respects the upper bound Δ from time GST onward.

This constraint might at first seem too strong: In realistic situations, the upper bound cannot reasonably be expected to hold forever after GST, but perhaps only for a limited time. However, any good solution to the consensus problem in this model would have an upper bound L on the amount of time after GST required for consensus to be reached; in this case it is not really necessary that the bound Δ hold forever after time GST, but only up to time GST + L. We find it technically convenient to avoid explicit mention of the interval length L in the model, but will instead present the appropriate upper bounds on time for each of our algorithms.

Instead of requiring that the consensus problem be solvable in the GST model, we might think of separating the correctness conditions into *safety* and *termination* properties. The safety conditions are that no two correct processors should ever reach disagreement, and that no correct processor should ever make a decision that is contrary to the specified validity conditions. The termination property is just that each correct processor should eventually make a decision. Then we might require an algorithm to satisfy the safety conditions no matter how asynchronously the message system behaves, that is, even if Δ does not hold eventually. On the other hand, we might only require termination in case Δ holds eventually. It is easy to see that these safety and termination conditions are equivalent to our GST condition: If an algorithm solves the consensus problem when Δ holds from time GST onward, then that algorithm cannot possibly violate a safety property even if the message system is completely asynchronous. This is because safety violations must occur at some finite point in time, and there would be some continuation of the violating execution in which Δ eventually holds.

Thus, the condition that Δ holds from some time GST onward provides a second reasonable definition for partial communication synchrony. Once again, it is not clear how we could apply previously known consensus protocols to this model. For example, the same argument as for the case of the unknown bound shows that we cannot treat lost or delayed messages in the same way as processor faults.

For succinctness, we say that communication is *partially synchronous* if one of these two situations holds: Δ exists but is not known, or Δ is known and has to hold from some unknown point on.

Our results determine precisely, for four interesting fault models, the maximum resiliency possible in cases where communication is partially synchronous. For fail-stop or omission faults we show that t-resilient consensus is possible iff $N \geq 2t + 1$. For Byzantine faults with authentication, we show that t-resilient consensus is possible iff $N \geq 3t + 1$. Also, for Byzantine faults without authenti-

TABLE I. SMALLEST NUMBER OF PROCESSORS N_{min} FOR WHICH A t-RESILIENT CONSENSUS PROTOCOL EXISTS

Failure type	Synchronous	Asynchronous	Partially synchronous communication and synchronous processors	Partially synchronous communication and processors	Partially synchronous processors and synchronous communication
Fail-stop	t	∞	$2t + 1$	$2t + 1$	t
Omission	t	∞	$2t + 1$	$2t + 1$	$[2t, 2t + 1]$
Authenticated Byzantine	t	∞	$3t + 1$	$3t + 1$	$2t + 1$
Byzantine	$3t + 1$	∞	$3t + 1$	$3t + 1$	$3t + 1$

cation, we show that t-resilient consensus is possible iff $N \geq 3t + 1$. (The "only if" direction in this case follows immediately from the result for the completely synchronous case in [15].) For all four types of faults, the time required for all correct processors to reach consensus is (1) a polynomial in N and Δ, for the model in which Δ is unknown; and (2) GST plus a polynomial in N and Δ, for the GST model. All of our protocols that reach consensus within time polynomial in parameters such as N, Δ, and GST also have the property that the total number of message bits sent is also bounded above by a polynomial in the same parameters.

Table I shows the maximum resiliency in various cases and compares our results with previous work. The results where communication is partially synchronous and processors are synchronous are shown in column 3 of the table; the results in columns 4 and 5 will be explained shortly. In each case, the table gives N_{min}, the smallest value of N ($N \geq 2$) for which there is a t-resilient protocol ($t \geq 1$). (Some of the lower bounds on N_{min} in the last column of the table have slightly stronger constraints on t and N, which are given in the formal statements of the theorems.) Results in the synchronous column are due to [3], [5], and [15], and those in the asynchronous column are due to [4] and [10]. The table entry that is the closed interval $[2t, 2t + 1]$ means that $2t \leq N_{min} \leq 2t + 1$.

It is interesting to note that, for fail-stop, omission, and Byzantine faults with authentication, the maximum resiliency for partially synchronous communication lies strictly between the maximum resiliency for the synchronous and asynchronous cases. It is also interesting to note that, for partially synchronous communication, authentication does not improve resiliency.

Our protocols use variations on a common method: A processor p tries to get other processors to change to some value v that p has found to be "acceptable"; p decides v if it receives sufficiently many acknowledgments from others that they have changed their value to v, so that a value different from v will never be found acceptable at a later time. Similar methods have already appeared in the literature (e.g., see [2], [19]). Reischuk [17] and Pinter [16] have also obtained consensus results that treat message and processor faults separately.

1.3 PARTIALLY SYNCHRONOUS COMMUNICATION AND PROCESSORS. It is easy to extend the models described in Section 1.2 to allow processors, as well as communication, to be partially synchronous. That is, Φ (the upper bound on relative processor speed) can exist but be unknown, or Φ can be known but actually hold only from some time GST onward. We obtain results that completely characterize the resiliency in cases in which both communication and processors are partially synchronous, for all four classes of faults. In such cases we assume that communication and processors possess the same type of partial synchrony; that is, either both Φ and Δ are unknown, or both hold from some time GST on.

Surprisingly, the bounds we obtain are exactly the same as for the case in which communication alone is partially synchronous; see column 4 of Table I. (The only difference is that in this case the polynomial bounds on time depend on N, Δ, and Φ.) In the earlier case the fact that Φ was equal to 1 implied that each processor could maintain a local time that was guaranteed to be perfectly synchronized with the local times of other processors. In this case no such notion of time is available. We give two new protocols allowing processors to simulate *distributed clocks*. (These are fault-tolerant variations on the clock used by Lamport in [13].) One uses $2t + 1$ processors and tolerates t fail-stop, omission, or authenticated Byzantine faults, while the other uses $3t + 1$ processors and tolerates t unauthenticated Byzantine faults. When the appropriate clock is combined with each of our protocols for the case where only communication is partially synchronous, the result is a new protocol for the case in which both communication and processors are partially synchronous.

1.4 PARTIALLY SYNCHRONOUS PROCESSORS. In analogy to our treatment of partial communication synchrony, it is easy to define models where processors are partially synchronous and communication is synchronous (Δ exists and is known a priori). The last column of Table I summarizes our results for this case. Once again, time is polynomial (this time in N, Δ and Φ). The basic strategy used in constructing the protocols for this case also involves combining a consensus protocol that assumes processor synchrony with a distributed clock protocol. For fail-stop faults and Byzantine faults with authentication, either the distributed clock or the consensus protocol can tolerate more failures than the corresponding clock or consensus protocol used for the case in which both communication and processors are partially synchronous, so we obtain better resiliencies.

Technical Remarks

(1) Our protocols assume that an atomic step of a processor is either to receive messages or to send a message to a single processor, but not both; there is neither an atomic receive/send operation nor an atomic broadcast operation. We adopt this rather weak definition of a processor's atomic step in this paper because it is realistic in practice and seems consistent with assumptions made in much of the previous work on distributed agreement. However, our lower bound arguments are still valid if a processor can receive messages and broadcast a message to all processors in a single atomic step.

(2) The strong unanimity condition requires that, if all initial values are the same, say v, then v must be the common decision. Weak unanimity requires this condition to hold only if no processor is faulty. Unless noted otherwise, our consensus protocols achieve strong unanimity, and our lower bounds hold even for weak unanimity. In the case, however, of Byzantine faults with authentication and partially synchronous processors, the upper bound $2t + 1$ in the last column of Table I holds for strong unanimity only if the initial values are signed by a distinguished "sender." This assumption is also used in the algorithm of [3] for the completely synchronous case. (For weak unanimity, the upper bound $2t + 1$ in the last column holds even without signed initial values.) We discuss this further in Section 6, which is the first place where the issue of whether the initial values are signed has any effect on our results.

(3) Our consensus protocols are designed for an arbitrary value domain V, whereas our lower bounds hold even for the case $|V| = 2$.

The remainder of this paper is organized as follows: Section 2 contains definitions. Section 3 contains our basic protocols, presented in a basic round model, which has more power than the models in which we are really interested. Section 4 contains our results for the model in which processors are synchronous and communication is partially synchronous. In particular, the protocols of Section 3 are adapted to this model. The distributed clocks are defined in Section 5, where we also discuss how to combine the results of Section 3 with the clocks to produce protocols for the model in which both processors and communication are partially synchronous. Section 6 contains our results for the case in which processors are partially synchronous and communication is synchronous.

2. *Definitions*

2.1 MODEL OF COMPUTATION. Our formal model of computation is based on the models of [4] and [10]. Here we review the basic features of the model informally. The communication system is modeled as a collection of N sets of messages, called *buffers*, one for each processor. The buffer of p_i represents messages that have been sent to p_i, but not yet received. Each processor follows a deterministic protocol involving the receipt and sending of messages. Each processor p_i can perform one of the following instructions in each *step* of its protocol:

Send(m, p_j): places message m in p_j's buffer;
Receive(p_i): removes some (possibly empty) set S of messages from p_i's buffer and delivers the messages to p_i.

In the Send(m, p_j) instruction, p_j can be any processor; that is, the communication network is completely connected. A processor's protocol is specified by a state transition diagram; the number of states can be infinite. The instruction to be executed next depends on the current state, and the execution causes a state transition. For a Send instruction, the next state depends only on the current state, whereas, for a Receive instruction, the next state depends also on the set S of delivered messages. The initial state of a processor p_i is determined by its initial value v_i in V. At some point in its computation, a processor can irreversibly *decide* on a value in V.

For subsequent definitions, it is useful to imagine that there is a *real-time clock* outside the system that measures time in discrete integer-numbered steps. At each tick of real time, some processors take one step of their protocols. A *run* of the system is described by specifying the initial states for all processors and by specifying, for each real-time step,

(1) which processors take steps,
(2) the instruction that each processor executes, and
(3) for each Receive instruction, the set of messages delivered.

Runs can be finite or infinite. Given an infinite run R, the message m is *lost* in run R if m is sent by some Send(m, p_j), p_j executes infinitely many Receive instructions in R, and m is never delivered by any Receive(p_j).

2.2 FAILURES. A processor *executes correctly* if it always performs instructions of its protocol (transition diagram) correctly. A processor is *correct* if it executes correctly and takes infinitely many steps in any infinite run. We consider four types of increasingly destructive faulty behavior of processor p_i:

Fail-stop: Processor p_i executes correctly, but can stop at any time. Once stopped it cannot restart.

Omission: Faulty processor p_i follows its protocol correctly, but Send(m, p_j), when executed by p_i, might not place m in p_j's buffer and Receive(p_i) might cause only a subset of the delivered messages to be actually received by p_i. In other words, an omission fault on reception occurs when some set S of messages is delivered to p_i and all messages in S are removed from p_i's buffer, but p_i follows a state transition as though some (possibly empty) subset S' of S were delivered.

Authenticated Byzantine: Arbitrary behavior, but messages can be signed with the name of the sending processor in such a way that this signature cannot be forged by any other processor.

Byzantine: Arbitrary behavior and no mechanism for signatures, but we assume that the receiver of a message knows the identity of the sender.

2.3 PARTIAL SYNCHRONY. Let I be an interval of real time and R be a run. We say that the communication bound Δ *holds in I for run R* provided that, if message m is placed in p_j's buffer by some Send(m, p_j) at a time s_1 in I, and if p_j executes a Receive(p_j) at a time s_2 in I with $s_2 \geq s_1 + \Delta$, then m must be delivered to p_j at time s_2 or earlier. This says intuitively that Δ is an upper bound on message transmission time in the interval I. The processor bound Φ *holds in I for R* provided that, in any contiguous subinterval of I containing Φ real-time steps, every correct processor must take at least one step. This implies that no correct processor can run more than Φ times slower than another in the interval I.

The following conditions, which define varying degrees of communication synchrony, place constraints on the kinds of runs that are allowed. In these definitions, Δ denotes some particular positive integer:

(1) Δ *is known*: The communication bound Δ holds in $[1, \infty)$ for every run R.
Delta is known: Δ is known for some fixed Δ. This is the usual definition of *synchronous communication.*

(2) *Delta is unknown*: For every run R, there is a Δ that holds in $[1, \infty)$.

(3) Δ *holds eventually*: For every run R, there is a time T such that Δ holds in $[T, \infty)$. Such a time T is called the *Global Stabilization Time* (*GST*).
Delta holds eventually: Δ holds eventually for some fixed Δ.

If either (2) or (3) holds, we say that *communication is partially synchronous.*

It is helpful to view each situation as a game between a protocol designer and an adversary. If delta is known, the adversary names an integer Δ, and the protocol designer must supply a consensus protocol that is correct if Δ always holds. If delta is unknown, the protocol designer supplies the consensus protocol first, then the adversary names a Δ, and the protocol must be correct if that Δ always holds. If delta holds eventually, the adversary picks Δ, the designer (knowing Δ) supplies a consensus protocol, and the adversary picks a time T when Δ must start holding.

By replacing Δ by Φ and "delta" by "phi" above, (1) defines *synchronous processors*, and (2) and (3) define two types of *partially synchronous processors.*

2.4 CORRECTNESS OF A CONSENSUS PROTOCOL. Given assumptions A about processor and communication synchrony, a fault type F, and a number N of processors and an integer t with $0 \leq t \leq N$, correctness of a *t-resilient consensus protocol* is defined as follows:

For any set C containing at least $N - t$ processors and any run R satisfying A and in which the processors in C are correct and the behavior of the processors not in C is allowed by the fault type F, the protocol achieves:

—*Consistency.* No two different processors in C decide differently.

—*Termination.* If R is infinite, then every processor in C makes a decision.

—*Unanimity.* There are two types:
Strong unanimity: If all initial values are v and if any processor in C decides, then it decides v.
Weak unanimity: If all initial values are v, if C contains all processors, and if any processor decides, then it decides v.

In models where messages cannot be lost, such as the models where delta is unknown, our protocols can be easily modified so that all correct processors can halt soon after sufficiently many correct processors have decided. However, we do not require halting explicitly in the termination condition because, as can be easily shown, if messages can be lost before GST in the model where delta holds eventually and if the protocol is 1-resilient to fail-stop faults, then there is some execution in which some correct processor does not halt. Further discussion of the issue of halting is given in Section 4.2, Remark 2, after the protocols have been described.

3. *The Basic Round Model*

In this section we define the basic round model and present preliminary versions of our algorithms in this model. In the following sections we show how each of our models can simulate the basic model.

3.1 Definition of the Model. In the basic round model, processing is divided into synchronous *rounds* of message exchange. Each round consists of a *Send subround*, a *Receive subround*, and a *computation subround*. In a Send subround, each processor sends messages to any subset of the processors. In a Receive subround, some subset of the messages sent to the processor during the corresponding Send subround is delivered. In a computation subround, each processor executes a state transition based on the set of messages just received. Not all messages that are sent need arrive; some can be lost. However, we assume that there is some round GST, such that all messages sent from correct processors to correct processors at round GST or afterward are delivered during the round at which they were sent. As explained in the Introduction, loss of a message *before* GST does not necessarily make the sender or the receiver faulty. Although all processors have a common numbering for the rounds, they do not know when round GST occurs. The various kinds of faults are defined for the basic model as for the earlier models.

3.2 Protocols in the Basic Round Model. In the remainder of this section, we show how the consensus problem can be solved for the basic model, for each of the fault types. To argue that our protocols achieve strong unanimity, we use the notion of a *proper* value defined as follows: If all processors start with the same value v, then v is the only proper value; if there are at least two different initial values, then all values in V are proper. In all protocols, each processor will maintain a local variable PROPER, which contains a set of values that the processor knows to be proper. Processors will always piggyback their current PROPER sets on all messages. The way of updating the PROPER sets will vary from algorithm to algorithm. If only weak unanimity is desired, the PROPER sets are not needed, and the protocols can be simplified somewhat; we leave these simplifications to the interested reader.

3.2.1 *Fail-Stop and Omission Faults.* The first algorithm is used for either fail-stop or omission faults. It achieves strong unanimity for an arbitrary value domain V.

Algorithm 1. $N \geq 2t + 1$

Initially, each processor's set PROPER contains just its own initial value. Each processor attaches its current value of PROPER to every message that it sends. Whenever a processor p receives a PROPER set from another processor that contains a particular value v, then p puts v into its own PROPER set. It is easy to check that each PROPER set always contains only proper values.

The rounds are organized into alternating *trying* and *lock-release* phases, where each trying phase consists of three rounds and each lock-release phase consists of one round. Each pair of corresponding phases is assigned an integer, starting with 1. We say that phase h *belongs to* processor p_i if $h \equiv i(\text{mod } N)$.

At various times during the algorithm, a processor may *lock* a value v. A *phase number* is associated with every lock. If p locks v with associated phase number $k \equiv i \bmod N$, it means that p thinks that processor p_i might decide v at phase k. Processor p only releases a lock if it learns its supposition was false. A value v is *acceptable* to p if p does not have a lock on any value except possibly v. Initially, no value is locked.

We now describe the processing during a particular trying phase k. Let $s = 4k - 3$ be the number of the first round in phase k, and assume $k \equiv i \bmod N$. At round s each processor (including p_i) sends a list of all its acceptable values that are also in its proper set to processor p_i (in the form of a (list, k) message). (If V is very large, it is more efficient to send a list of proper values and a list of unacceptable values. Given these lists, the proper acceptable values are easily deduced.) Just after round s, that is, during the computation subround between rounds s and $s + 1$, processor p_i attempts to choose a value to *propose*. In order for processor p_i to propose v, it must have heard that at least $N - t$ processors (possibly including itself) find value v acceptable and proper at the beginning of phase k. There might be more than one possible value that processor p_i might propose; in this case processor p_i will choose one arbitrarily. Processor p_i then broadcasts a message (lock v, k) at round $s + 1$.

If any processor receives a (lock v, k) message at round $s + 1$, it locks v, associating the phase number k with the lock, and sends an acknowledgment to processor p_i (in the form of an (ack, k) message), at round $s + 2$. In this case any earlier lock on v is released. (Any locks on other values are not released at this time.)

If processor p_i receives acknowledgments from at least $t + 1$ processors at round $s + 2$, then processor p_i decides v. After deciding v, processor p_i continues to participate in the algorithm.

Lock-release phase k occurs at round $s + 3 = 4k$. At round $s + 3$, each processor p broadcasts the message (v, h) for all v and h such that p has a lock on v with associated phase h. If any processor has a lock on some value v with associated phase h, and receives a message (w, h') with $w \neq v$ and $h' \geq h$, then the processor releases its lock on v.

LEMMA 3.1. *It is impossible for two distinct values to acquire locks with the same associated phase.*

PROOF. In order for two values v and w to acquire a lock at trying phase k, the processor to which phase k belongs must send conflicting (lock v, k) and (lock w, k) messages, which it will never do in this fault model. □

LEMMA 3.2. *Suppose that some processor decides v at phase k, and k is the smallest numbered phase at which a decision is made. Then at least $t + 1$ processors lock v at phase k. Moreover, each of the processors that locks v at phase k will, from that time onward, always have a lock on v with associated phase number at least k.*

PROOF. It is clear that at least $t + 1$ processors lock v at phase k. Assume that the second conclusion is false. Then let l be the first phase at which one of the locks on v set at phase k is released without immediately being replaced by another, higher numbered lock on v. In this case the lock is released during lock-release phase l, when it is learned that some processor has a lock on some $w \neq v$ with associated phase h, where $k \leq h \leq l$. Lemma 3.1 implies that no processor has a lock on any $w \neq v$ with associated phase k. Therefore, some processor has a lock on w with associated phase h, where $k < h \leq l$. Thus, it must be that w is found acceptable to at least $N - t$ processors at the first round of some phase numbered h, $k < h \leq l$, which means that at least $N - t$ processors do not have v locked at the beginning of that phase. Since $t + 1$ processors have v locked at least through the first round of l, this is impossible. □

LEMMA 3.3. *Immediately after any lock-release phase that occurs at or after GST, the set of values locked by correct processors contains at most one value.*

PROOF. Straightforward from the lock-release rule. □

THEOREM 3.1. *Assume the basic model with fail-stop or omission faults. Assume $N \geq 2t + 1$. Then Algorithm 1 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

PROOF. First, we show consistency. Suppose that some correct processor p_i decides v at phase k, and this is the smallest numbered phase at which a decision is made. Then Lemma 3.2 implies that, at all times after phase k, at least $t + 1$ processors have v locked. Consequently, at no later phase can any value other than v ever be acceptable to $N - t$ processors, so no processor will ever decide any value other than v.

Next, we argue strong unanimity. If all the initial values are v, then v is the only value that is ever in the PROPER set of any processor. Thus, v is the only possible decision value.

Finally, we argue termination. Consider any trying phase k belonging to a correct processor p_i that is executed after a lock-release phase, both occurring at or after round GST. We claim that processor p_i will reach a decision at trying phase k (if it has not done so already). By Lemma 3.3, there is at most one value locked by correct processors at the start of trying phase k. If there is such a locked value, v, then sufficient communication has occurred by the beginning of trying phase k so that v is in the PROPER set of each correct processor. Moreover, any initial value of a correct processor is in the PROPER set of each correct processor at the beginning of trying phase k. Since there are at least $N - t \geq t + 1$ correct processors, it follows that a proper, acceptable value will be found for processor p_i to propose, and that the proposed value will be decided on by processor p_i at trying phase k. □

It is easy to see that all correct processors make decisions by round GST + $4(N + 1)$.

3.2.2 *Byzantine Faults with Authentication.* The second algorithm achieves strong unanimity for an arbitrary value set V, in the case of Byzantine faults with authentication.

Algorithm 2. $N \geq 3t + 1$

Initially, each processor's PROPER set contains just its own initial value. Each processor attaches its PROPER set and its initial value to every message it sends. If a processor p ever receives $2t + 1$ initial values from different processors, among which there are not $t + 1$ with the same value, then p puts all of V (the total value domain) into its PROPER set. (Of course, p would actually just set a bit indicating that PROPER contains all of V.) When a processor p receives claims from at least $t + 1$ other processors that a particular value v is in their PROPER sets, then p puts v into its own PROPER set. It is not difficult to check that each PROPER set for a correct processor always contains only proper values.

Processing is again divided into alternating trying and lock-release phases, with phases numbered as before and of the same length as before. At various times during the algorithm, processors may lock values. In Algorithm 2, not only is a phase number associated with every lock, but also a *proof of acceptability* of the locked value, in the form of a set of signed messages, sent by $N - t$ processors, saying that the locked value is acceptable and in their PROPER sets at the beginning of the given phase. A value v is *acceptable* to p if p does not have a lock on any value except possibly v.

We now describe the processing during a particular trying phase k. Let $s = 4k - 3$ be the first round of phase k, and assume $k \equiv i \bmod N$. At round s, each processor p_j (including p_i) sends a list of all its acceptable values that are also in its PROPER set to processor p_i, in the form $E_j(\text{list}, k)$, where E_j is an authentication function. Just after round s, processor p_i attempts to choose a value to propose. In order for processor p_i to propose v, it must have heard that at least $N - t$ processors find value v acceptable and proper at phase k. Again, if there is more than one possible value that processor p_i might propose, then it will choose one arbitrarily. Processor p_i then broadcasts a message $E_i(\text{lock } v, k, \text{proof})$, where the proof consists of the set of signed messages $E_j(\text{list}, k)$ received from the $N - t$ processors that found v acceptable and proper.

If any processor receives an $E_i(\text{lock } v, k, \text{proof})$ message at round $s + 1$, it decodes the proof to check that $N - t$ processors find v acceptable and proper at phase k. If the proof is valid, it locks v, associating the phase number k and the message $E_i(\text{lock } v, k, \text{proof})$ with the lock, and sends an acknowledgment to processor p_i. In this case any earlier lock on v is released. (Any locks on other values are not released at this time.) If the processor should receive such messages for more than one value v, it handles each one similarly. The entire message $E_i(\text{lock } v, k, \text{proof})$ is said to be a *valid lock* on v at phase k.

If processor p_i receives acknowledgments from at least $2t + 1$ processors, then processor p_i decides v. After deciding v, processor p_i continues to participate in the algorithm.

Lock-release phase k occurs at round $s + 3 = 4k$. Processors broadcast messages of the form $E_i(\text{lock } v, h, \text{proof})$, indicating that the sender has a lock on v with associated phase h and the given associated proof, and that processor p_i sent the message at phase h, which caused the lock to be placed. If any processor has a lock on some value v with associated phase h and receives a properly signed message

E_j(lock w, h', proof′) with $w \neq v$ and $h' \geq h$, then the processor releases its lock on v.

LEMMA 3.4. *It is impossible for two distinct values to acquire valid locks at the same trying phase if that phase belongs to a correct processor.*

PROOF. In order for different values v and w to acquire valid locks at trying phase k, the processor p_i to which phase k belongs must send conflicting E_i(lock v, k, proof) and E_i(lock w, k, proof′) messages, which correct processors can never do. □

LEMMA 3.5. *Suppose that some correct processor decides v at phase k, and k is the smallest numbered phase at which a decision is made by a correct processor. Then at least $t + 1$ correct processors lock v at phase k. Moreover, each of the correct processors that locks v at phase k will, from that time onward, always have a lock on v with associated phase number at least k.*

PROOF. Since at least $2t + 1$ processors send an acknowledgment that they locked v at phase k, it is clear that at least $t + 1$ correct processors lock v at phase k. Assuming that the second conclusion is false, the remaining proof by contradiction is identical to the proof of Lemma 3.2. □

LEMMA 3.6. *Immediately after any lock-release phase that occurs at or after GST, the set of values locked by correct processors contains at most one value.*

PROOF. Straightforward from the lock-release rule. □

THEOREM 3.2. *Assume the basic model with Byzantine faults and authentication. Assume $N \geq 3t + 1$. Then Algorithm 2 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

PROOF. The proofs of consistency and strong unanimity are as in the proof of Theorem 3.1. To argue termination, consider any trying phase k belonging to a correct processor p_i that is executed after a lock-release phase, both occurring at or after GST. We claim that processor p_i will reach a decision at trying phase k (if it has not done so already). By Lemma 3.6, there is at most one value locked by correct processors at the start of trying phase k. If there is such a locked value v, then v was found to be proper to at least $N - t$ processors, of which $N - 2t \geq t + 1$ must be correct. Therefore, by the beginning of trying phase k, these $t + 1$ correct processors have communicated to all correct processors that v is proper, so by the way the set PROPER is augmented every correct processor will have v in its PROPER set by the beginning of trying phase k. Next, consider the case in which no value is locked at the beginning of trying phase k (so all values are acceptable). If there are at least $t + 1$ correct processors with the same initial value v, then v is in the PROPER set of each correct processor at the beginning of trying phase k. On the other hand, if this is not the case, then all values in the value set are in the PROPER set of all correct processors at the beginning of trying phase k. It follows that a proper, acceptable value will be found for processor p_i to propose, and that the proposed value will be decided on by processor p_i at trying phase k. □

As in the previous case, GST + $4(N + 1)$ is an upper bound on the number of rounds required for all the correct processors to reach decisions.

3.2.3 *Byzantine Faults without Authentication.* In this section we modify Algorithm 2 to handle Byzantine faults without authentication, while maintaining the same requirement, $N \geq 3t + 1$, on the number of processors and maintaining polynomial time complexity and polynomial message lengths. The modification is done by using a minor variation of a broadcast primitive, introduced by Srikanth and Toueg [20], which simulates the crucial properties of authentication. We first state these properties, then give the broadcast primitive, and finally describe the new agreement protocol.

The broadcast primitive (and hence the agreement algorithm that uses the broadcast primitive) is defined in terms of *superrounds*, where each superround consists of two normal Send–Receive rounds. Superround GST occurs at the earliest superround, when both of its Send–Receive rounds occur at or after round GST. The primitive gives an algorithm for a processor p to BROADCAST a message m at superround k and also gives conditions under which a processor will *accept* a message m from p (which is not to be confused with our definition of an "acceptable value"). The crucial properties of broadcasting that are used in the (authenticated) Algorithm 2 are as follows:

(1) *Correctness.* If a correct processor p BROADCASTS m in superround $k \geq$ GST, then every correct processor accepts m from p in superround k.
(2) *Unforgeability.* If a correct processor p does not BROADCAST m, then no correct processor ever accepts m from p.
(3) *Relay.* If a correct processor accepts m from p in superround r, then every other correct processor accepts m from p in superround max(r + 1, GST) or earlier.

The description of the BROADCAST primitive is given in Figure 1. The proof that the primitive has the Correctness and Unforgeability properties is identical to the proof of Srikanth and Toueg [20]. We give the proof of the Relay property since it is slightly different than in [20].

Say the correct processor q accepts m from p in superround r. Therefore, q must have received (echo, p, m, k) from at least $N - t$ processors by the end of the second round of superround r, so at least $N - 2t$ correct processors sent (echo, p, m, k). By definition of BROADCAST, if a correct processor sends (echo, p, m, k) at some round h, it continues to send (echo, p, m, k) at all rounds after h. Therefore, every correct processor will receive (echo, p, m, k) from at least $N - 2t$ processors by the end of the first round of superround max(r + 1, GST). Hence, every correct processor will send (echo, p, m, k) at the second round of max(r + 1, GST). So every correct processor will receive $N - t$ (echo, p, m, k) messages by superround max(r + 1, GST) or earlier, and will accept m from p.

The only difference between the protocol of Figure 1 and that of Srikanth and Toueg [20] is in the relaying of an (echo, p, m, k) message after $N - 2t$ (echo, p, m, k) messages have been received. In our case, the echo message continues to be sent at every round after the $N - 2t$ echoes are received, whereas in [20] the echo is sent only once. Since messages can be lost before GST in our model, the resending seems to be needed to get the Relay property. Although resending the echoes makes message length grow proportionally to the round number (since a new invocation of BROADCAST could be started at each round), message length is still polynomial in N and GST. In models where messages cannot be lost, such as the unknown delta model, each (echo, p, m, k) need be sent only once by each correct processor, resulting in shorter messages.

Next follows the new algorithm for the unauthenticated Byzantine case in the basic model. It is patterned after the authenticated algorithm. In particular, handling of PROPER sets is done exactly as in Algorithm 2.

BROADCAST of m by p at superround k: $N \geq 3t + 1$

Superround k:

First round: p sends (init, p, m, k) to all;

Second round: Each processor executes the following for any message m:

if received (init, p, m, k) from p in the first round and received only one init message from p in the first round, then send (echo, p, m, k) to all;
if received (echo, p, m, k) from at least $N - t$ distinct processors in this round, then accept m from p;

All subsequent rounds:
Each processor executes the following for any message m:

if received (echo, p, m, k) from at least $N - 2t$ distinct processors in previous rounds, then send (echo, p, m, k) to all;
if received (echo, p, m, k) from at least $N - t$ distinct processors in this or previous rounds, then accept m from p.

FIG. 1. The BROADCAST primitive.

Algorithm 3. $N \geq 3t + 1$

Processing is again divided into trying and lock-release phases, with phases numbered as before. Each trying phase takes three superrounds, that is, six ordinary rounds. Lock-release phase k is done during the third superround of trying phase k. As before, a value v is acceptable to p if p does not have a lock on any value except possibly v.

We now describe the processing during a particular trying phase k. Let $s = 3k - 2$ be the first superround of phase k, and assume $k \equiv i \bmod N$. At superround s, each processor p_j (including p_i) BROADCASTS a list of all its acceptable values that are also in its PROPER set in the form (list, k). Just after superround s, processor p_i attempts to choose a value to propose. In order for processor p_i to propose v, it must have accepted messages from at least $N - t$ processors stating that they find value v acceptable and proper at phase k. Again, if there is more than one possible value that processor p_i might propose, then it will choose one arbitrarily. Processor p_i then BROADCASTS a message (lock v, k) during superround $s + 1$.

If any processor q has by superround $s + 1$ accepted a message (lock v, k) from p_i and also accepted messages (list, k) from $N - t$ processors stating that they find v acceptable and proper at the first superround of phase k, then q locks v, associating the phase number k with the lock, and sends an acknowledgment (ack, k) to processor p_i. In this case any earlier lock on v is released. (Any locks on other values are not released at this time.) If the processor should receive such messages for more than one value v, it handles each one similarly. We say that *q accepts a valid lock on v with phase k* if it has accepted a message (lock v, k) from p_i and accepted $N - t$ messages (list, k) as just described. These messages do not all have to be accepted at the same round.

If processor p_i receives acknowledgments from at least $2t + 1$ processors, then processor p_i decides v. After deciding v, processor p_i continues to participate in the algorithm.

Lock-release phase k occurs at the end of the third superround of phase k. In this algorithm the lock-release phase does not send any messages. If a processor q has a lock on some value v with associated phase h and q has accepted, at this round or earlier, a valid lock on w with associated phase h', and if $w \neq v$ and $h' \geq h$, then q releases its lock on v.

LEMMA 3.7. *It is impossible for correct processors to accept valid locks on two distinct values with associated phase k if phase k belongs to a correct processor.*

PROOF. Suppose that the lemma is false. By the Unforgeability property, the processor p_i to which phase k belongs must BROADCAST conflicting (lock v, k) and (lock w, k) messages, which correct processors can never do. □

LEMMA 3.8. *Suppose that some correct processor decides v at phase k, and k is the smallest numbered phase at which a decision is made by a correct processor. Then at least t + 1 correct processors lock v at phase k. Moreover, each of the correct processors that locks v at phase k will, from that time onward, always have a lock on v with associated phase number at least k.*

PROOF. Since at least $2t + 1$ processors send an acknowledgment that they locked v at phase k, it is clear that at least $t + 1$ correct processors lock v at phase k. The rest of the proof is similar to the proofs of Lemmas 3.2 and 3.5, using the Unforgeability property to argue that, if a correct processor q accepts a valid lock on value $w \neq v$ with associated phase h, then w is found acceptable to all but at most t of the correct processors at the first round of phase h. □

LEMMA 3.9. *Immediately after any lock-release phase that occurs at or after GST, the set of values locked by correct processors contains at most one value.*

PROOF. Straightforward from the lock-release rule and the Relay property. □

THEOREM 3.3. *Assume the basic model with Byzantine faults without authentication. Assume $N \geq 3t + 1$. Then Algorithm 3 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

PROOF. The proof is virtually identical to the proof of Theorem 3.2, using the Correctness and Relay properties after GST to argue termination. □

An upper bound on the number of rounds required is GST + $6(N + 1)$.

Remark 1. Algorithms 1–3 have the property that all correct processors make a decision within $O(N)$ rounds after GST. The time to reach agreement after GST can be improved to $O(t)$ rounds by some simple modifications. The bound $O(t)$ is optimal to within a constant factor, since $t + 1$ rounds are necessary even if communication and processors are both synchronous and failures are fail-stop [7, 9]. A modification to all the algorithms is to have a processor repeatedly broadcast the message "Decide v" after it decides v. For Algorithm 1 (fail-stop and omission faults), a processor can decide v when it receives any "Decide v" message. For Algorithms 2 and 3 (Byzantine faults), a processor can decide v when it receives $t + 1$ "Decide v" messages from different sources. Easy arguments show that the modified algorithms are still correct and that all correct processors make a decision within $O(t)$ rounds after GST; these arguments are left to the reader.

4. *Partially Synchronous Communication and Synchronous Processors*

In this section we assume that processors are completely synchronous ($\Phi = 1$) and communication is partially synchronous. We show how to use these models to simulate the basic model of Section 3.1 and thus to solve the same consensus problem.

Since processors operate in lock-step synchrony, it is useful to imagine that each (correct) processor has a clock that is perfectly synchronized with the clocks of other correct processors. Initially, the clock is 0, and a processor increments its clock by 1 every time it takes a step. The assumption $\Phi = 1$ implies that the clocks of all correct processors are exactly the same at any real-time step.

As presented in Section 2, there are two different definitions of partially synchronous communication: (1) delta is unknown, and (2) delta holds eventually. We consider these two cases separately. Section 4.1 describes the upper bound results for the model in which delta holds eventually. Section 4.2 describes the upper bound results for delta unknown. Finally, Section 4.3 contains the lower bound results.

4.1 UPPER BOUNDS WHEN DELTA HOLDS EVENTUALLY. We first consider the model in which delta holds eventually. Fix any of the four possible fault models. We show that, if there is a t-resilient consensus protocol in the basic model, then there is one in the model in which delta holds eventually. To see the implication, fix Δ and assume algorithm A works for the basic model. From A we define an algorithm A' for the model in which Δ holds eventually.

Let $R = N + \Delta$. Each processor divides its steps into groups of R, and uses each group to simulate its own actions in a single round of algorithm A. More specifically, the processor uses the first N steps of group r to send its round r messages to the N processors, sending to one processor at a time, and uses the last Δ steps to perform Receive operations. The state transition for round r is simulated at the last step of group r. (The number R is large enough to allow all processors to exchange messages within a single group of steps, once GST has been reached.) Each processor always attaches a round identifier (number) to messages, and any message sent during a round r that arrives late during some round $r' > r$ is ignored. Thus, communication during each round is independent of communication during any other round.

For any run e' of A', it is easy to show that there exists a corresponding run e of A with the following properties:

(1) All processors that are correct in e' are also correct in e.
(2) The types of faults exhibited by the faulty processors are the same in e' as in e.
(3) Every state transition of a correct processor in e is simulated by the corresponding correct processor in e'.

Since algorithm A is assumed to be a t-resilient consensus protocol for the basic round model, consensus is eventually reached in e, and so in e', as needed.

By applying the transformation just described to Algorithms 1–3, we obtain Algorithms 1^1–3^1, respectively. We immediately obtain the following result:

THEOREM 4.1. *Assume that processors are completely synchronous* ($\Phi = 1$) *and communication is partially synchronous* (Δ *holds eventually*).

(*a*) *For the fail-stop or omission fault model, if* $N \geq 2t + 1$, *then Algorithm* 1^1 *achieves consistency, strong unanimity, and termination for an arbitrary value domain.*
(*b*) *For the authenticated Byzantine fault model, if* $N \geq 3t + 1$, *then Algorithm* 2^1 *achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

(*c*) *For the unauthenticated Byzantine fault model, if* $N \geq 3t + 1$, *then Algorithm* 3^1 *achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

It is easy to see that Algorithms 1^1 and 2^1 guarantee that decisions are reached by all correct processors within time $4(N+1)(N+\Delta)$ after GST. The corresponding bound for Algorithm 3^1 is $6(N+1)(N+\Delta)$. Thus, the time for Algorithms 1^1–3^1 is bounded above by GST plus a polynomial in N and Δ. Remark 1 at the end of Section 3 shows how these time bounds can be improved. As mentioned in the Introduction, these bounds also give the time after GST when Δ can stop holding again.

4.2 UPPER BOUNDS FOR DELTA UNKNOWN. Now we consider the model in which delta is unknown. Fix any of the four possible fault models. We show that, if there is a t-resilient protocol in the basic model, then there is one in the model in which delta is unknown. Let algorithm A work for the basic model. As before, we define A' from A so that every execution of A' is a simulation of an execution of A.

Let $R_r = N + r$. Each processor in A' divides its steps into groups so that its rth group contains exactly R_r steps. As before, the processor uses each group to simulate its own actions in a single round of algorithm A. Thus, the processor uses the first N steps of group r to send its round r messages to the N processors, one processor at a time, and uses the last r steps to perform Receive operations. The round r state transition is simulated at the last step of group r. Again, each processor always attaches a round identifier (number) to messages, and any message sent during a round r that arrives late during some round $r' > r$ is ignored.

Now consider any run e' of A', and assume that the communication bound Δ holds in e'. As before, it is easy to define a corresponding run e of A. The number of steps in e' that are allotted for the simulation of any round $r \geq \Delta$ is sufficient to allow all messages that are sent during round r to get received. Thus, e is an allowable run of A (with Δ as its GST round). Since A is assumed to be a t-resilient consensus protocol for the basic model, consensus is eventually reached in e, and so in e', as needed.

By applying this transformation to Algorithms 1–3, we obtain Algorithms 1^2–3^2, respectively, and immediately obtain the following result:

THEOREM 4.2. *In the model in which processors are completely synchronous* ($\Phi = 1$) *and communication is partially synchronous* (*delta is unknown*), *claims* (*a*)–(*c*) *of Theorem* 4.1 *hold for Algorithms* 1^2–3^2, *respectively.*

We now bound the time required by Algorithms 1^2–3^2. Consider Algorithm 1^2, for example, and fix any execution e with corresponding message bound Δ. Then round Δ is the GST for the execution of Algorithm 1 simulated by e. It requires at most time $\Delta(N + \Delta)$ for processors to complete their simulations of the first Δ rounds of Algorithm 1 (Δ rounds, with $N + \Delta$ as the maximum time to simulate a single round). Then an additional $4(N + 1)$ rounds, at most, must be simulated. These additional rounds require at most time $4(N+1)(N+\Delta+4(N+1))$, where the term $(N + \Delta + 4(N + 1))$ represents the maximum time to simulate one of these rounds (the last and largest one). Thus the total time is bounded by $\Delta(N + \Delta) + 4(N + 1)(N + \Delta + 4(N + 1))$, or $O(N^2 + \Delta^2)$. The same bound holds for Algorithm 2^2. The corresponding bound for Algorithm 3^2 is $\Delta(N + \Delta) + 6(N+1)(N+\Delta+6(N+1))$. Thus the time for Algorithms 1^2–3^2 is bounded above by a polynomial in N and Δ. Again, these bounds can be improved using the ideas in Remark 1 at the end of Section 3.

Remark 2. If we strengthen the model where delta holds eventually to require that no messages are ever lost, but that messages sent before GST can arrive late, then we can modify Algorithms 1^1–3^1 to allow processors to terminate. Specifically, we use the ideas described in Remark 1 at the end of Section 3. In the present case, however, each processor need only broadcast a single "Decide v" message, at the time when it decides v. This message is not tagged with a round number, and other processors should accept a "Decide v" message at any time. For fail-stop or omission faults, a processor can stop participating in the algorithm immediately after it broadcasts its "Decide v" message. Further, it can decide v immediately after receiving a "Decide v" message. For Byzantine faults, a processor can decide v after receiving $t + 1$ "Decide v" messages, but it cannot stop participating in the algorithm until after it has broadcast its "Decide v" message and received "Decide v" messages from a total of $2t + 1$ processors. If messages can be lost before GST, it is not hard to argue that, in any consensus protocol resilient to one fail-stop fault, there is some execution in which at least one correct processor must continue sending messages forever. The argument is similar to those for Theorems 4.3 and 4.4 in the next subsection. All that is needed to ensure halting in practice, however, is that each correct processor be able to reliably deliver a "Decide v" message to every other correct processor; in the absence of network partition, this could be done by repeated sending.

Remark 3. All the results of this section have assumed $\Phi = 1$. If processors are synchronous with $\Phi > 1$ and communication is partially synchronous, we would hope to obtain the same results. We show that this extension holds by proving a more general set of results: In Section 5 we show that the resiliency achieved by the protocols of this section can also be achieved if both processors and communication are partially synchronous. These stronger results imply that the same resiliency is achievable if communication is partially synchronous and processors are synchronous with $\Phi > 1$.

4.3 LOWER BOUNDS. In this section we give our lower bound results for partially synchronous communication and completely synchronous processors. The first lower bound shows that the resiliency of Theorems 4.1 and 4.2, part (a), cannot be improved, even for weak unanimity and a binary value domain.

THEOREM 4.3. *Assume the model with fail-stop or omission faults, where the processors are synchronous and communication is partially synchronous (either delta holds eventually or delta is unknown). Assume* $2 \le N \le 2t$. *Then there is no t-resilient consensus protocol that achieves weak unanimity for binary values.*

PROOF. The proof is the same for both definitions of partially synchronous communication. Assume the contrary, that there is an algorithm immune to fail-stop faults satisfying the required properties. We shall derive a contradiction.

Divide the processors into two groups, P and Q, each with at least 1 and at most t processors. First consider the following Scenario A: All initial values are 0, the processors in Q are initially dead, and all messages sent from processors in P to processors in P are delivered in exactly time 1. By t-resiliency, the processors in P must reach a decision; say that this occurs within time T_A. The decision must be 0. For if it were 1, we could modify the scenario to one in which the processors in Q are alive but all messages sent from Q to P take more than time T_A to be delivered. In the modified scenario, the processors in P still decide 1, contradicting weak unanimity.

Consider Scenario B: All initial values are 1, the processors in P are initially dead, and messages sent from Q to Q are delivered in exactly time 1. By a similar argument, the processors in Q decide 1 within T_B steps for some finite T_B.

Consider Scenario C (for Contradiction): Processors in P have initial values 0, processors in Q have initial values 1, all processors are alive, messages sent from P to P or from Q to Q are delivered in exactly time 1, and messages sent from P to Q or from Q to P take more than $\max(T_A, T_B)$ steps to be delivered. The processors in group P (respectively, group Q) act exactly as they do in Scenario A (respectively, Scenario B). This yields a contradiction. □

The following lower bound result again applies in the case of weak unanimity and a binary value domain. It shows that the resiliency of Theorems 4.1 and 4.2, part (b), cannot be improved, even for the case of weak unanimity and a binary value domain.

THEOREM 4.4. *Assume the model with Byzantine faults and authentication, in which the processors are synchronous and communication is partially synchronous (either delta holds eventually or delta is unknown). Assume $2 \leq N \leq 3t$. Then there is no t-resilient consensus protocol that achieves weak unanimity for binary values.*

PROOF. Again, the proof is the same for both definitions of partially synchronous communication. Assume the contrary. We shall derive a contradiction.

If $N = 2$, then the theorem follows from the previous lower bound, Theorem 4.3. Assume then that $N \geq 3$. Divide the processors into three groups, P, Q, and R, each with at least 1 and at most t processors. First consider the following Scenario A: All initial values are 0, the processors in R are initially dead, and all messages sent from processors in $P \cup Q$ to processors in $P \cup Q$ are delivered in exactly time 1. By t-resiliency, the processors in $P \cup Q$ must reach a decision; say that this occurs within time T_A. As in the previous lower bound proof, the decision must be 0.

Consider Scenario B: All initial values are 1, the processors in P are initially dead, and messages sent from $Q \cup R$ to $Q \cup R$ are delivered in exactly time 1. By a similar argument, the processors in $Q \cup R$ decide 1 within T_B steps for some finite T_B.

Consider Scenario C: Processors in P have initial values 0, processors in R have initial values 1, and processors in Q are faulty. The processors in Q behave with respect to those in P exactly as they do in Scenario A, and with respect to those in R exactly as they do in Scenario B. The messages sent from P to $P \cup Q$ and from R to $R \cup Q$ are delivered in exactly time 1, but all messages from P to R or from R to P take more than $\max(T_A, T_B)$ steps to be delivered. The processors in group P (respectively, group R) act exactly as they do in Scenario A (respectively, Scenario B). This yields a contradiction. □

The preceding lower bound is tight for the case of unauthenticated Byzantine faults (Theorems 4.1 and 4.2, part (c)).

5. *Partially Synchronous Communication and Processors*

In this section we consider the case in which both communication and processors are partially synchronous. We show the existence of protocols with the same resiliencies as in the previous section, where only communication was partially synchronous. Moreover, the algorithms for corresponding cases still require amounts of time (specifically, polynomial) similar to the earlier case. Again, we proceed by showing how to use the models of this section to simulate the basic model of Section 3.

In the previous section, the processors had a common notion of time that allowed time to be divided into rounds. In this case, where phi does not always hold or is unknown, no such common notion of time is available. Therefore, our first task is to describe protocols that give the processors some approximately common notion of time. We call such protocols *distributed clocks.*

Our distributed clocks do not use explicit knowledge of Δ or Φ. They are designed to be used in either kind of partially synchronous model, delta and phi holding eventually or delta and phi unknown. However, the properties that the clocks exhibit do depend on the particular bounds Δ and Φ that hold (eventually) during the particular run.

Each processor maintains a private (software) clock. The private clocks grow at a rate that is within some constant factor of real time and remain within a constant of each other. For the model with delta and phi unknown, these conditions hold at all times. For the GST model, however, these conditions are only guaranteed to hold after some constant amount of time after GST. The three "constants" here depend polynomially on N, Φ, and Δ. We have made no effort to optimize these constants, as this would obfuscate an already difficult and technical argument. In addition, the number of message bits sent by correct processors is polynomially bounded in N, Δ, Φ, and GST.

Once we have defined the distributed clocks, the protocols of Section 3 are simulated by letting each processor use its private clock to determine which round it is in. Several "ticks" of each private clock are used for the simulation of each round in the basic model. In order to use a distributed clock in such simulations, we need to interleave the steps of the distributed-clock algorithm with steps belonging to the underlying algorithm being simulated. Moreover, the distributed-clock algorithm itself is conveniently described as interleaving Receive steps, which increase the recipient's knowledge of other processors' local clocks, with Send steps, which allow the sender to inform others about its local clock. To be specific, we assume that processors alternately execute a Receive operation for the clock, a Send operation for the clock, and a step of the algorithm being simulated.

In this section we describe what happens during the clock maintenance steps for two different distributed clocks. The first, presented in Section 5.1, handles Byzantine faults without authentication and requires $N \geq 3t + 1$. The second, presented in Section 5.2, handles Byzantine faults with authentication and requires $N \geq 2t + 1$. This clock obviously handles fail-stop and omission faults as well. In Section 5.3 the upper bounds for the model in which delta and phi hold eventually are given. In Section 5.4 we present the upper bound results for the model in which delta and phi are unknown. We do not prove lower bounds in this section, since the lower bounds obtained in Section 4 apply to the current models.

5.1 A DISTRIBUTED CLOCK FOR BYZANTINE FAULTS WITHOUT AUTHENTICATION. Throughout this section we assume that $N \geq 3t + 1$. We again assume that real times are numbered 0, 1, 2, ... Processors participate in our distributed clock protocols by sending ticks to one another. As an expositional convenience, we define a master clock whose value at any time s depends on the past global behavior of the system and is a function of the ticks that have been sent before s. Even approximating the value of the master clock requires global information about what ticks have been sent to which processors. We therefore introduce a second type of message, called a *claim*, in which processors make assertions about the ticks they have sent.

An *i-tick* is the message i. An *i^+-tick* is a j-tick for any $j \geq i$. We say p has *broadcast an i-tick* if it has sent an i^+-tick to all N processors.

An *i-claim* is the message "I have broadcast an i-tick." An *i^+-claim* is a j-claim for any $j \geq i$. We say p has *broadcast an i-claim* if it has sent an i^+-claim to all N processors.

We adopt the convention that all processors have exchanged ticks and claims of size 0 before time 0. These messages are not actually sent, but they are considered to have been sent and received. When we say that a certain event, such as the receipt of a certain message, has occurred "by time s," we mean that the event has occurred at some real-time step $\leq s$.

The master clock, $C: \mathbf{N} \to \mathbf{N}$, is defined at any real time s by

$C(s)$ = maximum j such that $t + 1$ correct processors have broadcast a j-tick by time s.

Since all processors are assumed by convention to have broadcast a 0-tick before time 0, $C(0) = 0$. Note that $C(s)$ is a nondecreasing function of s.

For each processor p_i, the private clock, $c_i: \mathbf{N} \to \mathbf{N}$, is defined by

$c_i(s)$ = maximum j such that, by time s, p_i has received either
(1) messages from $2t + 1$ processors, where each message is a j^+-claim, or
(2) messages from $t + 1$ processors, where each message is either a $(j + 1)^+$-tick or a $(j + 1)^+$-claim.

Since p_i is assumed to have received 0-claims from all N processors before time 0, $c_i(0) = 0$ for all correct p_i. Note that $c_i(s)$ is nondecreasing for all correct p_i.

Let p_i be a correct processor. In sending ticks, p_i's goal is to increment the master clock, so ideally we would like p_i to send a $(C(s) + 1)$-tick at time s. However, knowing $C(s)$ requires global information. Instead, p_i uses c_i, its view of C, to compute its next tick, sending a $(c_i(s) + 1)$-tick at time s. We show in Lemma 5.1 that $c_i(s) \leq C(s)$, so p_i will never force the master clock to skip a value. We also show that, "soon" after GST for the GST model, the value of the master clock exceeds those of the private clocks by at most a constant amount, so that p_i will not be pushing the master clock far ahead of the private clocks of the other processors.

Each processor p_i repeatedly cycles through all N processors, broadcasting, in different cycles, either ticks or claims. The private clock of p_i is stored in a local variable c_i. Processor p_i updates its private clock every time it executes a Receive operation in the clock protocol by considering all the ticks and claims it has received and updating its private clock according to the definition of the private clock given above (thus the private clock is updated every second clock step, i.e., every third step, that p_i takes). The following two programs describe how ticks and claims are sent during the sending steps of the clock protocol. A processor begins the distributed clock protocol by setting c_i to 0 and calling TICK(0), where TICK(b) is the protocol shown in Figure 2. Note that the value of c_i may change during an execution of TICK(b), but only a $(b + 1)$-claim (rather than a $(c_i + 1)$-claim) is sent during execution of CLAIM(b). This is consistent with our definition of what it means to have broadcast a $(b + 1)$-tick.

```
TICK(b):
  for j = 1, . . . , N do
    send (c_i + 1)-tick to p_j;
  CLAIM(b).
CLAIM(b):
  for j = 1, . . . , N do
    send (b + 1)-claim to p_j;
  if c_i > b then TICK(c_i) else CLAIM(b).
```

FIG. 2. The TICK and CLAIM procedures.

The following lemmas describe limitations on the rates of the master clock and the local clocks. The first three lemmas do not involve Δ and Φ, and so apply to either partially synchronous model (delta and phi holding eventually or delta and phi unknown).

LEMMA 5.1. *For all $s \geq 0$ and for all i such that p_i is correct, $c_i(s) \leq C(s)$.*

PROOF. The proof is by induction on s. The basis $s = 0$ is obvious since $c_i(0) = C(0) = 0$ by definition.

Fix some s and some correct p_i, and assume that the statement of the lemma is true for all $s' < s$ and all correct p_k. Let $j = c_i(s)$. By the definition of the private clock, there are two possibilities:

(1) p_i has received j^+-claims from $2t + 1$ different processors. Since at least $t + 1$ of these j^+-claims are from correct processors, $C(s) \geq j$ by definition of the master clock.

(2) p_i has received messages from $t + 1$ different processors, each of which is either $(j + 1)^+$-tick or a $(j + 1)^+$-claim. Consider the earliest real time, s', when some correct processor, say p_k, sends a $(j + 1)^+$-tick. Note that $s' < s$, so $c_k(s') \leq C(s')$ by the inductive hypothesis. By definition of the protocol, $c_k(s') \geq j$. Therefore,

$$j \leq c_k(s') \leq C(s') \leq C(s). \qquad \square$$

LEMMA 5.2. *For all $s \geq 0$, the largest tick sent by a correct processor at real time s has size at most $C(s) + 1$.*

PROOF. This proof is immediate from the protocol and Lemma 5.1. $\square$

LEMMA 5.3. *For all $s, x \geq 0$, $C(s + x) \leq C(s) + x$.*

PROOF. The proof is by induction on x. For the basis, let $x = 1$. By Lemma 5.2 the largest tick sent by a correct processor by time s has size at most $C(s) + 1$, so the maximum tick that can be broadcast by $t + 1$ processors by time $s + 1$ is a $(C(s) + 1)$-tick. Thus, $C(s + 1) \leq C(s) + 1$. Assume the lemma holds for some x. Then

$$\begin{aligned} C(s + (x + 1)) = C((s + 1) + x) &\leq C(s + 1) + x \\ &\qquad \text{(by the induction hypothesis)} \\ &\leq C(s) + (x + 1) \qquad \text{(by the basis).} \end{aligned} \qquad \square$$

The preceding lemmas are independent of both communication and processor synchrony. Now we give several lemmas that assume such synchrony. We would like to state the lemmas in a way that applies to both kinds of partially synchronous models (delta and phi holding eventually and delta and phi unknown). So fix Δ and Φ (for either case). Also fix GST for the model in which Δ and Φ hold eventually. For the model in which delta and phi are unknown, define GST = 0, for uniformity.

The next few lemmas discuss the behavior of the clocks a short time after GST. Lemma 5.4 says that the private clocks increase at most a constant factor more slowly than real time. Lemmas 5.5 and 5.6 are technical lemmas used to prove the following lemma. Lemma 5.7 has two parts: The first says that, at any particular real time, the master clock exceeds the value of the private clocks by at most an additive constant. The second part of Lemma 5.7 says that the master clock runs at a rate at most a constant factor slower than real time.

Let $D = \Delta + 3\Phi$. Note that, if a message is sent to a correct processor p at time $s \geq$ GST, then p will receive the message by time $s + D$: The message will be delivered by time $s + \Delta$, and within an additional time 3Φ, p will execute a Receive operation in the clock protocol.

LEMMA 5.4. *Assume $s \geq GST$, and let $s' = s + 12N\Phi + D$. Let j be such that $c_i(s) \geq j$ for all correct p_i. Then $c_i(s') \geq j + 1$ for all correct p_i.*

PROOF. At time s, p_i could be executing TICK(b) for some $b < j$. However, within time $6N\Phi$ after s, p_i will call TICK(b') or CLAIM(b') for some $b' \geq j$, and within an additional $6N\Phi$ steps, p_i will broadcast a $(j + 1)$-claim. Therefore, every correct processor will broadcast a $(j + 1)$-claim by time $s + 12N\Phi$. By time s', each correct p_i will receive at least $2t + 1 (j + 1)^+$-claims, so $c_i(s') \geq j + 1$. □

LEMMA 5.5. *Assume $s \geq GST$, and let $s' = s + 39N\Phi + 4D$. Then $C(s') \geq C(s) + 2$.*

PROOF. Let $j = C(s)$. By definition of the master clock, $t + 1$ correct processors have broadcast a j-tick by time s. These $t + 1$ processors send a tick or claim of size at least j to every processor within the first $3N\Phi$ steps after time s. Since these messages are sent after GST, they are received within D steps, so $c_i(s + 3N\Phi + D) \geq j - 1$ for all correct p_i. By three applications of Lemma 5.4, $c_i(s') \geq j + 2$. So $C(s') \geq j + 2$ by Lemma 5.1. □

LEMMA 5.6. *Let s_0 be the minimum time such that $C(s_0) \geq C(GST) + 2$. (Time s_0 exists by Lemma 5.5.) Let $s \geq s_0 + D$. Then $c_i(s) \geq C(s - D) - 1$ for all correct p_i.*

PROOF. Let $j = C(s - D)$. Then $t + 1$ correct processors broadcast a j-tick by $s - D$. By Lemma 5.2, the largest tick sent by a correct processor by GST is a $(C(\text{GST}) + 1)$-tick. Since $j \geq C(\text{GST}) + 2$, the j-ticks from correct processors are broadcast entirely after GST, so they are received by time s. Thus, for all correct p_i, $c_i(s) \geq j - 1$. □

LEMMA 5.7. *Let s_0 be the minimum time such that $C(s_0) \geq C(GST) + 2$.*

(*a*) *For all $s \geq s_0 + D$ and for all correct processors p_i, $c_i(s) \geq C(s) - D - 1$.*
(*b*) *For all $s \geq s_0$ and for $s' = s + 24N\Phi + 3D$, $C(s') \geq C(s) + 1$.*

PROOF

(a) Lemma 5.6 implies $c_i(s) \geq C(s - D) - 1$. By Lemma 5.3, $C(s) \leq C(s - D) + D \leq c_i(s) + 1 + D$. Thus, $c_i(s) \geq C(s) - D - 1$.
(b) Let $x = s + D$. Lemma 5.6 implies $c_i(x) \geq C(s) - 1$ for all correct p_i. By two applications of Lemma 5.4, $c_i(s') \geq C(s) + 1$. So $C(s') \geq C(s) + 1$ by Lemma 5.1. □

5.2 A DISTRIBUTED CLOCK FOR BYZANTINE FAULTS WITH AUTHENTICATION. The new clock is very similar to the one just described. We only explain the differences. Here we assume $N \geq 2t + 1$.

An *i-claim* is a signed message "I have broadcast an i-tick." An *i^+-claim* is a j-claim for any $j \geq i$. For $i \geq 1$, an *i-tick* is the message "(i, i-proof)," where a 1-*proof* is the empty string and where an *i-proof* ($i > 1$) is a list of $t + 1 (i - 1)^+$-claims each signed by a different processor. An *i^+-tick* is a j-tick for any $j \geq i$. The definitions of *broadcast an i-tick* and *broadcast an i-claim* are the same as before.

The master clock C: $\mathbf{N} \rightarrow \mathbf{N}$ is defined by

$C(s)$ = maximum j such that some correct processor has broadcast a j-tick by time s.

The private clock c_i: $\mathbf{N} \rightarrow \mathbf{N}$ is defined by

$c_i(s)$ = maximum j such that p_i has received $t + 1$ j^+-claims (from different sources), either directly, or indirectly as part of a tick, by time s.

The definition of the clock protocol is the same as before with the addition that, whenever a processor sends a $(b + 1)$-claim in the procedure CLAIM(b), it attaches the largest size tick that it can construct (this will always be a $(b + 1)^+$-tick). A correct processor will ignore any received j-claim if it does not come with an attached j^+-tick. The reason for this modification is so that correct processors will not accept claims that are much too large from faulty processors and incorporate these large claims into proofs.

LEMMA 5.8. *Lemmas* 5.1–5.7 *hold for the authenticated Byzantine clock.*

PROOF. The proofs are virtually identical to the proofs for the unauthenticated Byzantine clock, and most details are omitted. The major differences are the following:

The proof of Lemma 5.1 is easier since there is only one case. Letting $j = c_i(s)$, processor p_i has received $t + 1$ j^+-claims from different processors, at least one of which must be correct. Since a correct processor sends a j^+-claim only after it has broadcast a j-tick, we have $C(s) \geq j$ by definition of the master clock.

The proofs of Lemmas 5.2 and 5.3 are unchanged.

In the proof of Lemma 5.4, change "$2t + 1$" to "$t + 1$."

In the proof of Lemma 5.5, letting $j = C(s)$, we can only say that at least one correct processor has broadcast a j-tick by time s. However, this j-tick contains a j-proof consisting of $t + 1$ $(j - 1)^+$-claims, so we can conclude that $c_i(s + 3N\Phi + D) \geq j - 1$ for all correct p_i as before. The proof of Lemma 5.6 is changed similarly.

The proof of Lemma 5.7 follows from previous lemmas by calculations and is unchanged. □

We need one more lemma to support our claim that the number of message bits sent by correct processors is bounded above by a polynomial in GST, N, Δ, and Φ.

LEMMA 5.9. *For all $s \geq 0$, the largest tick sent by any processor (correct or faulty) at real time s has size at most $C(s) + 2$.*

PROOF. A j-tick sent at time s contains $t + 1$ $(j - 1)^+$-claims, at least one of which was sent by a correct processor. The conclusion now follows from Lemma 5.2. □

From this lemma and the definition of the protocol, it follows easily that any tick or claim sent by a correct processor at time s can be encoded in $O(t \log C(s))$ bits.

Remark 4. The clocks of Sections 5.1 and 5.2 are similar to the one discovered independently by Attiya et al. [1].

5.3 UPPER BOUNDS WHEN DELTA AND PHI HOLD EVENTUALLY. We now present our upper bound results for partially synchronous communication and processors, for the model where delta and phi hold eventually. Fix any of the four possible fault models. We show that, if there is a t-resilient protocol in the basic model, then there is one in the model where delta and phi hold eventually. To see the implication, fix Δ and Φ, and assume algorithm A works for the basic model. We define A' from A as follows, so that A' works for the model where Δ and Φ hold after GST.

As described above, two out of every three steps of each processor are used to maintain a distributed clock, and the other step is used to simulate algorithm A. For fail-stop or omission faults, we use the authenticated Byzantine clock, simplified appropriately because the signatures are not needed and because we cannot assume the authentication capability. Note that the consensus protocol and distributed clock protocol have the same constraint on the number of processors, $N \geq 2t + 1$. For unauthenticated Byzantine faults, we use the unauthenticated Byzantine clock. For authenticated Byzantine faults, either clock could be used.

The Receive steps of algorithm A' are designated as belonging to either the clock simulation or the algorithm simulation. However, each time a Receive step of A' occurs, it is possible that messages for either or both simulations will be received. We assume that each processor maintains a pair of message buffers, one for each of the two simulations it is carrying out. When the processor does a Receive step that belongs to the clock simulation, it saves any messages for the algorithm simulation in the algorithm message buffer, and vice versa. Also, each time the processor does a Receive step that belongs to the clock simulation, it collects not only the new incoming messages, but all those in the clock message buffer, to use in its clock simulation step; analogous assumptions are made for the algorithm simulation.

Fix $R = 3N\Phi + 2D + 2$, where, as before, $D = \Delta + 3\Phi$. Each processor uses its private clock to determine the round of algorithm A currently being simulated. Namely, if $(r - 1)R \leq c_i(s) < rR$, then processor p_i determines at real time s that the current round is r. Processors label messages with round numbers. As long as a processor determines that the current round is r, it uses its protocol simulation steps to simulate steps of round r in the basic model. The first N protocol simulation steps are used for sending the round r messages to all the processors, and the remaining steps are spent executing Receive operations. Unlike the simulations in Section 4, it is possible that there will be insufficient time for a processor to actually send all its round r messages.

Processor p_i simulates its state transition for round r at its first algorithm simulation step at which it decides the current round is strictly greater than r. More specifically, assume that processor p_i has reached an algorithm simulation step s, at which the current round is k, and assume that the round at processor p_i's last algorithm simulation step was $h < k$. Then processor p_i simulates its state transitions for rounds $h, h + 1, \ldots, k - 1$, all at the beginning of step s. In simulating these state transitions, processor p_i simulates all of its sending steps for these rounds; that is, it makes the appropriate state transitions, but does not actually send any messages, and it simulates the receipt of all the messages that are in the algorithm message buffer.

For any run e' of A', it is easy to define a corresponding run e of A. We see that all processors that are correct in e' are also correct in e, and that the types of faults exhibited by the faulty processors are the same in both cases. We argue that, within a short time after GST, the number of ticks in e' that are allotted for the simulation of any round r is sufficient to allow all round r messages to be sent and received. More precisely, the "short time after GST" is chosen so that parts (a) and (b) of Lemma 5.7 hold.

We must first show that there is sufficient time for each correct processor p_i to send all its round r messages and then to do at least one Receive operation. Assume that s is the first real time at which processor p_i's private clock reaches or exceeds $(r-1)R$. Then processor p_i would finish sending all its round r messages and doing one Receive operation by real time $s + 3(N+1)\Phi$. We must show that processor p_i's clock up to real time $s + 3(N+1)\Phi$ remains less than rR, that is, that

$$c_i(s + 3(N+1)\Phi) < rR. \tag{5.1}$$

We must also show that there is sufficient time for all round r messages sent by processor p_i to be received. Fix a correct processor p_j. We show that processor p_j has sufficient time to receive a round r message from processor p_i before going on to simulate round $r+1$. Again, letting s be the first real time for which $c_i(s) \geq (r-1)R$, p_i will send the message to p_j by real time $s + 3N\Phi$, and p_j will receive the message by real time $s + 3N\Phi + D$. Therefore, we must show that

$$c_j(s + 3N\Phi + D) < rR. \tag{5.2}$$

Since $D \geq 3\Phi$ and since clocks are nondecreasing, we can prove both (5.1) and (5.2) by showing that, for any correct processor p_k,

$$c_k(s + 3N\Phi + D) < rR.$$

This follows because

$$\begin{aligned}
c_k(s + 3N\Phi + D) &\leq C(s + 3N\Phi + D) && \text{(by Lemma 5.1)}\\
&\leq C(s-1) + 3N\Phi + D + 1 && \text{(by Lemma 5.3)}\\
&\leq c_i(s-1) + 3N\Phi + 2D + 2 && \text{(by Lemma 5.7(a))}\\
&< (r-1)R + 3N\Phi + 2D + 2 && \text{(by assumption)}\\
&= rR.
\end{aligned}$$

Since A is assumed to be a t-resilient consensus protocol for the basic model, consensus is eventually reached in e, and so in e', as needed.

By applying the transformation just described to Algorithms 1–3, we obtain Algorithms 1^3–3^3, respectively. We immediately obtain the following result:

THEOREM 5.1. *Assume that communication and processors are partially synchronous* (*delta and phi hold eventually*).

(*a*) *For the fail-step or omission fault model, if* $N \geq 2t + 1$, *then Algorithm* 1^3 *achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

(*b*) *For the authenticated Byzantine fault model, if* $N \geq 3t + 1$, *then Algorithm* 2^3 *achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

(c) *For the unauthenticated Byzantine fault model, if $N \geq 3t + 1$, then Algorithm 3^3 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

As before, we claim that Algorithms 1^3–3^3 reach agreement within a polynomial (in N, Δ, and Φ) amount of time after GST. Our claims of polynomial-time performance follow from the fact that the master clock, a short time after GST, runs at a rate no slower than $1/(24N\Phi + 3(\Delta + 3\Phi))$ times real time (see Lemma 5.7(b)). Finally, the total number of message bits sent by correct processors is polynomially bounded in N, Δ, Φ, and GST, since the number of bits in each message sent by a correct processor is polynomially bounded in these quantities.

5.4 UPPER BOUNDS WHEN DELTA AND PHI ARE UNKNOWN. Next, we present our upper bound results for partially synchronous communication and processors, for the model where delta and phi are unknown. The ideas are a simple combination of ideas from Sections 4.2 and 5.3. The transformation of a consensus protocol for the basic model to one for the model where delta and phi are unknown is identical to the transformation described in Section 5.3 except that the bound $R_r = 3Nr + 8r + 2$ is used to describe the number of ticks to be used for the simulation of round r. (This bound is obtained from the previous bound by replacing both Δ and Φ by r.) The proof of correctness is the same as before, since GST is reached when r exceeds the (unknown) Δ and Φ that hold in the run. By applying this transformation to Algorithms 1–3, we obtain Algorithms 1^4–3^4, respectively.

THEOREM 5.2. *Assume that communication and processors are partially synchronous (delta and phi are unknown). Then claims (a)–(c) of Theorem 5.1 hold for Algorithms 1^4–3^4, respectively.*

As before, it is easy to see that Algorithms 1^4–3^4 reach agreement within a polynomial (in N, Δ, and Φ) amount of time.

Remark 5. In the simulation of the basic model described in Sections 5.3 and 5.4, if the round number of processor p_i's last algorithm simulation step was h and processor p_i updates its clock and finds that it is now simulating some round $k > h$, then all state transitions in rounds h through $k - 1$ are simulated (except that no messages are sent). For a general simulation of the basic model, these transitions must all be simulated, since they may involve state transitions that processor p_i must make in order that the simulation of the algorithm in the basic model be correct. However, it is not hard to see that, for the particular Algorithms 1–3 designed for the basic model in Section 3.2, processor p_i can just simulate the state transition for round h and continue the simulation at round k, without simulating the "missed" transitions in rounds $h + 1$ through $k - 1$. This can be done since the state information in Algorithms 1–3 (not including the current round number) consists of the PROPER sets, the values which are locked, and other information associated with each lock. Changes in this state information are caused only by the receipt of certain messages. Since we have shown consistency for Algorithms 1–3 even if messages are lost before GST, it follows that the algorithms remain consistent if processors, including correct ones, skip state transitions before GST.

6. *Partially Synchronous Processors and Synchronous Communication*

In this section we consider models where processors are partially synchronous and communication is synchronous; that is, there is a fixed upper bound Δ on message transmission time that always holds (in particular, no messages are lost). Of course, the protocols of the previous section with their associated resiliencies work for such models, but by using the fact that communication is now synchronous, we can achieve higher resiliencies in some cases.

It is convenient to base our consensus algorithms on another basic model, which we call the *basic model with signals*. In Section 6.1 we define this new basic model and give consensus algorithms that are designed to work in the basic model with signals. We then show how to use the eventual phi and unknown phi models to simulate the basic model with signals. As in Section 5, we use distributed clocks to give the processors some approximately common notion of time. The clocks are discussed in Section 6.2. Section 6.3 contains algorithms for the case in which phi holds eventually, and Section 6.4 contains algorithms for phi unknown. Section 6.5 contains lower bounds.

6.1 A BASIC MODEL WITH SIGNALS. The basic model with signals is just like the basic model, except that the Receive subround also includes the possible receipt of a *signal* by each processor. In any round r, the receipt of a signal by processor p_i implies that all correct processors receive the round r messages sent to them by processor p_i. The nonreceipt of a signal does not imply anything. At round GST and afterward, we assume that all correct processors receive signals at each round. The next two subsections, 6.1.1 and 6.1.2, give consensus protocols for the basic model with signals that are resilient to two types of faults.

6.1.1 *Fail-Stop Faults.* The next algorithm achieves strong unanimity for an arbitrary value domain V.

Algorithm 4. $N \geq t$

Each processor has a local variable VALUE, initialized at its initial value. We say that each round $k \equiv i \bmod N$ *belongs to* processor p_i. Processing in an arbitrary round k is as follows:

Processing for p_i, where round k belongs to p_i:
Broadcast VALUE;
If a signal is received, then decide on VALUE.

Processing for p_j, where round k does not belong to p_j:
If a message is received with contents v, then set VALUE := v.

LEMMA 6.1. *Assume that processor p_i decides v at round k, and that this is the smallest numbered round at which a decision is made. Then no message containing value $w \neq v$ is ever sent at any round $\geq k$.*

PROOF. Assume for the sake of contradiction that the lemma is false, and let h be the smallest numbered round $\geq k$ when a message containing value $w \neq v$ is sent. It is clear that $h \neq k$, since faults are fail-stop. Let p_j be the processor that owns round h.

Since processor p_i receives a signal at round k, it must be the case that processor p_j receives value v from processor p_i at round k and therefore sets its VALUE to v. By assumption, no message with value different from v is sent at rounds after k and before h. Therefore, processor p_j's VALUE remains equal to v until the beginning of round h. This contradicts the assumption that processor p_j sends w at round h. □

THEOREM 6.1. *Assume the basic model with signals, with fail-stop faults. Assume $N \geq t$. Then Algorithm 4 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

PROOF. First, we show consistency. Suppose that some correct processor p_i decides v at round k, and this is the smallest numbered round at which a decision is made. Then Lemma 6.1 implies that no message containing value $w \neq v$ is ever sent at any round $\geq k$. But a processor can decide on a value w only if it first sends out messages containing w. Therefore, no processor ever decides on a value $w \neq v$.

Strong unanimity is obvious, since a message with contents v is only sent if v was the initial value of some processor.

Since a signal is received by each correct processor at every round on or after GST, by definition of the basic model with signals, it is obvious that each round on or after GST results in a decision for its owner if that owner has not already decided. □

6.1.2 *Authenticated Byzantine Faults.* The next algorithm, Algorithm 5, achieves weak unanimity for an arbitrary value domain.

Algorithm 5. $N \geq 2t + 1$

The protocol is similar to Algorithm 2 of Section 3.2.2, with a few changes as indicated below. Because we are only dealing with weak unanimity, the PROPER sets are not used. This time, the rounds are divided into trying phases of two rounds each and lock-release phases of one round each. A trying phase of Algorithm 5 is the same as the first two rounds of the corresponding trying phase of Algorithm 2, except that, if a processor, during one of its trying phases, is choosing a value to propose and if several values are acceptable, the processor chooses its own initial value if that value is acceptable or chooses arbitrarily otherwise. The third round is omitted; processor p_i does not wait for messages from others claiming that they have responded to a message E_i(lock v, k, proof) by locking v. Instead, it checks that a signal has been received at the second round of the trying phase. If a signal is received, then processor p_i decides v.

In Algorithm 2, processor p_i needed at least $2t + 1$ acknowledgment messages to conclude that at least $t + 1$ correct processors actually locked v at phase k. Now we can argue that, if a signal is received, then all correct processors will have actually locked v at phase k, and since $N \geq 2t + 1$, there are at least $t + 1$ correct processors.

The proof of the following theorem is very similar to that of Theorem 3.2 (the result about Algorithm 2), and details are left to the reader.

THEOREM 6.2. *Assume the basic model with signals, with authenticated Byzantine faults. Assume $N \geq 2t + 1$. Then Algorithm 5 achieves consistency, weak unanimity, and termination for an arbitrary value domain.*

One version of the consensus problem studied in the literature supposes that a distinguished processor, called the "general," gives the initial values v_i to all processors. In the case of Byzantine faults with authentication, it is usually assumed that the general signs these initial values with its own unforgeable signature. Thus, if the general is correct, there is a single value v such that the general gives a signed v to every processor; in this case, strong unanimity requires that v is the value decided by all correct processors. If the general is faulty, the general can give out different values and can even give two different values, both signed, to the same processor; in this case, strong unanimity does not require any particular value to be the decision value. This issue was not raised earlier because it is irrelevant to the results of Sections 3–5; that is, our protocols for the authenticated Byzantine case are designed to work even if the general does not sign the initial values, and our lower bound Theorem 4.4 is still valid if the general does sign the initial values. (If the general does sign the initial values, updating to the PROPER sets in Algorithm 2 can be simplified.) This distinction is important in the completely synchronous case: N-resilient strong unanimity is possible in the authenticated Byzantine case (column 1, row 3 of Table I) only if the general signs the initial values.

This distinction also matters in this section of the paper. Consider the basic model with signals, with authenticated Byzantine faults, where the general signs the initial values and where $N \geq 2t + 1$. Then a slight variant of Algorithm 5 achieves consistency, *strong* unanimity, and termination for an arbitrary value domain.

Algorithm 6. $N \geq 2t + 1$

The algorithm is identical to Algorithm 5, except that PROPER sets are used. Initially, the PROPER set of processor p_i contains its initial value v_i, which is signed by the general. Each processor piggybacks its initial value, signed by the general, on all messages. If p_i ever receives a value different from v_i that is also signed by the general, then p_i puts all of V in its PROPER set. It is clear that a correct processor's PROPER set always contains proper values.

6.2 DISTRIBUTED CLOCKS. Recall that in this section there is some known communication bound Δ that always holds. Because the previous clocks have limited resiliency, we first describe a distributed clock that is resilient to any number of fail-stop faults. The general form of the clock is similar to the clocks of Sections 5.1 and 5.2.

As in Section 5.1, an *i-tick* is the message i, and an *i-claim* is the message "I have broadcast an i-tick." The definitions of *i^+-tick*, *i^+-claim*, *broadcast an i-tick*, and *broadcast an i-claim* are also the same as in Section 5.1. The clock protocol is given by TICK(b) and CLAIM(b), as in Figure 2.

The master clock is

$C(s)$ = maximum j such that some processor has broadcast a j-tick by time s.

The private clock c_i is

$c_i(s)$ = maximum j such that p_i has received either a j^+-claim or a $(j + 1)^+$-tick by time s.

We claim that the new fail-stop clock and the authenticated Byzantine clock of Section 5.2, when used in the model of this section, have the following properties:

(A1) For all s and all correct p_i, $c_i(s) \leq C(s)$.

(A2) For all $s, x \geq 0$, $C(s + x) \leq C(s) + x$.

(A3) Consider a run in which the processor bound Φ holds after time GST (in the unknown phi model, GST = 0 for uniformity as explained before), and let $D = 3\Phi + \Delta$. There are constants a_1 and a_2 depending polynomially on N, Δ, and Φ such that

(A3.1) for all correct p_i and all $s \geq \text{GST} + a_1$, $c_i(s) \geq C(s) - D - 1$;
(A3.2) for all $s \geq \text{GST} + a_1$, $C(s + a_2) \geq C(s) + 1$.

(A4) For all s at which the correct processor p_i executes a Receive operation in the clock protocol,

$$C(s) - (\Delta + 1) \leq c_i(s).$$

To be technically precise, in the fail-stop case in Section 6 we consider a processor to be "correct" up until the time it fails (assuming that it does fail). In particular, the four properties above hold for all processors up until the time they fail.

For the authenticated Byzantine clock, we have already proved (A1), (A2), and (A3) in Lemmas 5.1, 5.3, and 5.7, respectively, with modifications as described in the proof of Lemma 5.8. To prove that these properties hold for the fail-stop clock, we first note that Lemmas 5.1–5.4 hold for the fail-stop clock; the proofs are very similar to the proofs given in Section 5.1 and are left to the reader. Lemma 5.5 is not needed. Since Δ always holds, we can prove a stronger version of Lemma 5.6 for the fail-stop clock.

LEMMA 5.6′. *For all $s \geq GST + D$ and all correct p_i, $c_i(s) \geq C(s - D) - 1$.*

PROOF. Let $j = C(s - D)$. By definition of the master clock, some processor has broadcast a j-tick by time $s - D$, so every correct processor will receive a j^+-tick by time s. Therefore, $c_i(s) \geq j - 1$, by definition of the private clock. □

Now Lemma 5.7 follows from Lemma 5.6′ and previous lemmas as before. (However, we only need $s \geq$ GST + D for part (a) and $s \geq$ GST for part (b)).

The proof of (A4) is similar for both clocks. Let $j = C(s - \Delta)$. A j-tick has been broadcast by time $s - \Delta$, so processor p_i, by time s, will receive a j^+-tick. For the authenticated Byzantine clock, this j^+-tick contains $t + 1$ $(j - 1)^+$-claims. For either clock, by definition of the private clock and by property (A2),

$$c_i(s) \geq j - 1 = C(s - \Delta) - 1 \geq C(s) - \Delta - 1.$$

6.3 UPPER BOUNDS WHEN PHI HOLDS EVENTUALLY. The only improvements over the case in which both phi and delta hold eventually are for fail-stop faults and authenticated Byzantine faults (the latter either for weak unanimity or for strong unanimity, with a general signing the initial values). Fix one of these fault models. We show that, if there is a t-resilient protocol in the basic model with signals, then there is one in the model where phi holds eventually. Fix Δ and Φ, and assume algorithm A works for the basic model with signals. Define A' as follows.

Two out of every three steps of each processor are used to maintain a distributed clock, and the other step is used to simulate algorithm A. For fail-stop faults, we use the new fail-stop clock of Section 6.2, while for authenticated Byzantine faults we use the authenticated Byzantine clock. Message buffers are maintained as in Section 5.3.

Fix $R = 3N\Phi + (2D + 2) + (\Delta + 1)$, where, as before $D = \Delta + 3\Phi$. Each processor determines the current round being simulated and conducts the rest of the simulation exactly as in Section 5.3. We must describe how signals are simulated. If a processor p_i has sent all its messages for a particular round r, performed a Receive operation in the clock protocol, and updated its private clock, and if the clock then satisfies

$$c_i < rR - (2\Delta + 1),$$

then p_i acts in A' as p_i would act in A if it had received a signal for round r.

For any run e' of A', we define a corresponding run e of A. Again, faults are preserved. Since the R in this section is larger than the R used in Section 5.3, it follows as in Section 5.3 that, within a short time after GST, the number of ticks in e' that are allotted for the simulation of any round r is sufficient to allow all round r messages to be sent and received. It remains to show that signals behave correctly:

(a) Whenever a correct processor p_i receives a signal at any round r, it means that all of the messages sent by processor p_i at round r to correct processors actually get received.
(b) Within a short time after GST, all correct processors receive signals at all rounds.

We first show (a). Assume that correct processor p_i receives a signal at round r, that p_i sends a message to correct processor p_j at round r, and that s is the real time when the message is sent. Then the message arrives at processor p_j by real time $s + \Delta$. Processor p_j might not actually receive the message at this time, since it is not executing a Receive operation at this time. However, the key fact for the simulation is that the message will be received the next time that p_j executes a Receive operation, and that, when this Receive occurs, p_j has not yet started any round greater than r. That is, we must show that

$$c_j(s + \Delta) < rR.$$

To show this, first note that, since processor p_i receives a signal for round r, there must be a real time s' with $s' > s$ such that p_i executes a Receive operation in the clock protocol at time s' and

$$c_i(s') < rR - (2\Delta + 1).$$

Now,

$$\begin{aligned}
c_j(s+\Delta) &\le C(s+\Delta) && \text{(by (A1))}\\
&\le C(s'+\Delta) && \text{(since } s' > s\text{)}\\
&\le C(s') + \Delta && \text{(by (A2))}\\
&\le c_i(s') + 2\Delta + 1 && \text{(by (A4))}\\
&< rR && \text{(by the condition defining simulation of signaling).}
\end{aligned}$$

Next, we show (b). Fix some round r after GST, and let s be the earliest time at which p_i's private clock reaches or exceeds $(r-1)R$. Processor p_i can broadcast a message to all processors and execute a Receive operation in the clock protocol within $3(N+1)\Phi$ steps after s. Therefore, we must show that

$$c_i(s + 3(N+1)\Phi) < rR - (2\Delta + 1).$$

This is true because

$$\begin{aligned}
c_i(s + 3(N+1)\Phi) &\le C(s + 3(N+1)\Phi) && \text{(by (A1))}\\
&\le C(s-1) + 3(N+1)\Phi + 1 && \text{(by (A2))}\\
&\le c_i(s-1) + 3(N+1)\Phi + D + 2 && \text{(by (A3.1))}\\
&< (r-1)R + 3(N+1)\Phi + D + 2 && \text{(by assumption)}\\
&\le rR - (2\Delta + 1) && \text{(by calculation).}
\end{aligned}$$

By applying this transformation to Algorithms 4, 5, and 6, we obtain Algorithms 4^1, 5^1, and 6^1, respectively.

THEOREM 6.3. *Assume that communication is synchronous and processors are partially synchronous* (*phi holds eventually*).

(*a*) *For the fail-stop model, if $N \geq t$, then Algorithm 4^1 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

(*b*) *For Byzantine faults with authentication, if $N \geq 2t + 1$, then Algorithm 5^1 achieves consistency, weak unanimity, and termination for an arbitrary value domain.*

(*c*) *For Byzantine faults with authentication, if $N \geq 2t + 1$ and if the general signs the initial values, then Algorithm 6^1 achieves consistency, strong unanimity, and termination for an arbitrary value domain.*

6.4 UPPER BOUNDS FOR PHI UNKNOWN. The strategy is the same as in Sections 4.2 and 5.4. Namely, we use the algorithm of Section 6.3 where $R_r = 3Nr + 6r + 3\Delta + 3$ steps are allowed for the simulation of round r, where R_r is obtained from the R of Section 6.3 by replacing Φ by r. It is important to note that the verification of (a) in Section 6.3 (viz., that if a signal is received by p_i at round r, then all messages sent by p_i during round r to correct processors arrive before the other processor starts any round greater than r) did not depend in any way on Φ. Therefore, (a) holds even for rounds r, where r is smaller than the actual (unknown) Φ that holds in the run. Applying this transformation to Algorithms 4, 5, and 6, we obtain Algorithms 4^2, 5^2, and 6^2, respectively.

THEOREM 6.4. *Assume that communication is synchronous and processors are partially synchronous (phi is unknown). Then claims (a), (b), and (c) of Theorem 6.3 hold for Algorithms 4^2, 5^2, and 6^2, respectively.*

Our claim of a polynomial time bound (after GST) for the algorithms of Sections 6.3 and 6.4 follows from clock property (A3.2), which states that the master clock runs fast enough after GST.

We should also mention that Remark 5 at the end of Section 5 does not apply to the simulations of Sections 6.3 and 6.4. Here, if a processor's clock makes a big jump so that rounds are missed, all steps of the consensus protocol during the missed round(s) must be simulated. If the correct p_i sends a message to a correct p_j and receives a signal during round r, then p_j must receive the message and make the appropriate state transition caused by this reception, even if p_j's clock makes a large jump that causes it to miss round r.

6.5 LOWER BOUNDS. The following lower bound shows that the resiliency of Theorems 6.3 and 6.4, parts (b) and (c), cannot be improved. The method used to prove this lower bound was suggested by Dolev (personal communication).

THEOREM 6.5. *Assume the model with Byzantine faults with authentication, synchronous communication, and partially synchronous processors. Assume $4 \leq N \leq 2t$. Then there is no t-resilient consensus protocol that achieves weak unanimity for binary values, even if the general signs the initial values.*

PROOF. Assume, to the contrary, that a consensus algorithm exists. The proof is identical for both variations of partially synchronous processors. In the following we assume, without loss of generality, that all messages are delivered in one real-time step. Divide the processors into four groups P, Q, $\{b\}$, and $\{r\}$, where groups P and Q each contain at least 1 and at most $t - 1$ processors and where b and r are single processors. We say that a processor *wakes up* at real time s if it takes the first step of its protocol at real time s. We say that a processor *runs fast* in the real-time interval $[s_1, s_2]$ if it takes a step of its protocol at each real-time step in the interval.

Consider Scenario CP, where the processors in $P \cup \{b\}$ have initial values 0, wake up at time 1, and run fast in the interval $[1, \infty)$, and where the other processors are initially dead. By t-resiliency, the processors in P make some decision within some finite time T_P. We claim the decision must be 0. For if it were 1, we could modify the scenario to one in which all initial values are 0, and the processors in $Q \cup \{r\}$ are correct but do not wake up until after time T_P. The processors in P still decide 1 in the modified scenario, which contradicts weak unanimity.

Consider the analogous Scenario CQ where the processors in $P \cup \{r\}$ are initially dead, and the processors in $Q \cup \{b\}$ wake up at time 1 with initial values 1 and run fast in the interval $[1, \infty)$. Therefore, the processors in Q decide 1 after some finite time T_Q.

Consider the following Scenario BP: Processors in $P \cup \{b\}$ are Byzantine. The processors in P have value 0, and b has both 0 and 1 (so the general is Byzantine). They wake up at time 1, with b acting as if its value is 0, and they run fast in the interval $[1, T_P]$. They send the same messages to r as are sent in Scenario CP, but no messages are sent to Q. After time T_P, the processors in P die. The processors in Q are correct. They wake up at time $T_P + 1$ and run fast thereafter. Starting at time $T_P + 1$, the Byzantine processor b starts behaving toward Q and r exactly as it does in Scenario CQ, as if its value were 1, except that a message sent at real time s in Scenario CQ is sent at time $T_P + s$ in Scenario BP. Since Q has received no messages from P, the processors in Q decide 1 at time $T_P + T_Q$, and they all behave exactly as in Scenario CQ, except that everything happens T_P real-time steps later. At time $T_P + T_Q + 1$, the correct processor r wakes up and runs fast thereafter. The initial value of r is irrelevant. Note that at most t processors are faulty in this run. In the model where phi is unknown, the processor bound $\Phi = T_P + T_Q + 1$ holds in this run; in the model where phi holds eventually, the processor bound $\Phi = 1$ holds after GST $= T_P + T_Q + 1$. Since the correct processors in Q have already decided 1 before r wakes up, r must decide 1 at some real time T_r.

Consider now Scenario BQ: The processors in P are correct and begin with value 0. They run fast in the interval $[1, T_P]$ but take no more steps until after time T_r. In the time interval $[1, T_P]$, the Byzantine processor b behaves toward P and r exactly as it does in Scenario CP, acting as if it had initial value 0. Therefore, at time T_P the processors in P decide 0. The processors in Q are Byzantine. They wake up at time $T_P + 1$ with value 1 and behave with respect to r exactly as they do in Scenario BP; that is, the messages that have been sent from P to Q during the interval $[1, T_P]$ are ignored by Q. At time $T_P + 1$, b starts acting toward r exactly as it does in Scenario BP, as if it had initial value 1. The correct processor r wakes up at time $T_P + T_Q + 1$ and runs fast thereafter. It is easy to see that the messages received by r between time $T_P + T_Q + 1$ and time T_r are exactly the same in Scenario BQ as in Scenario BP. Therefore, r decides 1 at time T_r, which is a contradiction because the correct processors in P decided 0. □

In the preceding proof, note that the processors in P and Q exhibit only omission faults: P fails to send messages to Q in Scenario BP, and Q fails to receive messages from P in Scenario BQ. Processor b is the only one that exhibits Byzantine behavior stronger than omission faults. Therefore, it can be checked that the same proof can be carried out for omission faults with three groups of processors, P, Q, and $\{r\}$, where P and Q each contain at least 1 and at most $t - 1$ processors. This proves the following, which shows that the resiliency of Theorems 5.1 and 5.2, part (a), when applied to the case of omission faults and partially synchronous processors, cannot be improved by more than 1.

THEOREM 6.6. *Assume the model with omission faults, synchronous communication, and partially synchronous processors. Assume* $3 \leq N \leq 2t - 1$. *Then there is no t-resilient consensus algorithm that achieves weak unanimity for binary values.*

For the case of strong unanimity and Byzantine faults with authentication, but where the initial values are not signed by a general, Theorems 5.1 and 5.2, part (b), give consensus algorithms if $N \geq 3t + 1$. The following shows that this resiliency is the best possible for this case.

THEOREM 6.7. *Assume the model with Byzantine faults with authentication, synchronous communication, and partially synchronous processors. Assume* $3 \leq N \leq 3t$. *If the general does not sign the initial values, there is no t-resilient consensus protocol that achieves strong unanimity for binary values.*

PROOF. Assume $N \leq 3t$. Divide the processors into three groups, P, Q, and R, each containing at least 1 and at most t processors.

Consider the following Scenario A: Processors in P have initial values 0, processors in Q have initial values 1, processors in $P \cup Q$ wake up at time 1 and run fast thereafter, and processors in R are initially dead. Therefore, the processors in $P \cup Q$ must make some decision after some finite time. By symmetry we can assume, without loss of generality, that they decide 1 within time T_A.

Consider Scenario B: All processors have initial values 0, processors in R are correct but do not wake up until after time T_A, and processors in Q are Byzantine and behave with respect to P exactly as they do in Scenario A. The processors in group P act exactly as they do in Scenario A, so they decide 1. This contradicts strong unanimity. □

7. *Open questions*

(1) We have noted in Remark 1 at the end of Section 3 that the basic consensus Algorithms 1–3, with minor modifications, have the property that the number of rounds required to reach agreement after round GST is optimal to within constant factors (at most 12). We have not tried to reduce these constants. Some reduction is probably possible, say by overlapping trying phases with lock-release phases, although it would be surprising if the number of rounds could be made to match the known lower bound of $t + 1$ rounds. On the other hand, partial synchrony might provide a model for which the lower bound $t + 1$ could be strengthened to something larger.

(2) A general direction for future research is to study other distributed computing problems in partially synchronous models.

ACKNOWLEDGMENTS. Joe Halpern asked whether the impossibility results of [4] and [10] would continue to hold in case the parameters Φ or Δ exist but are not known a priori, and this led to the formulation of the version of partial synchrony where phi or delta are unknown. We are grateful to Jennifer Lundelius Welch who read a draft of this paper and provided many helpful comments.

REFERENCES

1. ATTIYA, A., DOLEV, D., AND GIL, J. Asynchronous Byzantine consensus. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29). ACM, New York, 1984, pp. 119–133.
2. BRACHA, G., AND TOUEG, S. Asynchronous consensus and broadcast protocols. *J. ACM 32*, 4 (Oct. 1985), 824–840.

3. DOLEV, D., AND STRONG, H. R. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput. 12* (1983), 656–666.
4. DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan. 1987), 77–97.
5. DOLEV, D., FISCHER, M. J., FOWLER, R., LYNCH, N. A., AND STRONG, H. R. Efficient Byzantine agreement without authentication. *Inf. Control 52* (1982), 257–274.
6. DOLEV, D., LYNCH, N. A., PINTER, S. S., STARK, E. W., AND WEIHL, W. E. Reaching approximate agreement in the presence of faults. *J. ACM 33*, 3 (July 1986), 499–516.
7. DWORK, C., AND MOSES, Y. Knowledge and common knowledge in a Byzantine environment I: Crash failures. In *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning about Knowledge* (Monterey, Calif., Mar. 19–22). Kaufmann, Los Altos, Calif., 1986, pp. 149–169.
8. FISCHER, M. J. The consensus problem in unreliable distributed systems (a brief survey). Rep. YALEU/DSC/RR-273. Dept. of Computer Science, Yale Univ., New Haven, Conn., June 1983.
9. FISCHER, M. J., AND LAMPORT, L. Byzantine generals and transaction commit protocols. Tech. Rep. Op. 62, SRI International, Menlo Park, Calif., 1982.
10. FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.
11. GARCIA-MOLINA, H., PITTELLI, F., AND DAVIDSON, S. Is Byzantine agreement useful in a distributed database? In *Proceedings of the 3rd SIGACT-SIGMOD Symposium on Principles of Database Systems* (Waterloo, Ont., Canada, Apr. 2–4). ACM, New York, 1984, pp. 61–69.
12. GRAY, J. N. Notes on database operating systems. In *Operating Systems: An Advanced Course*. Lecture Notes in Computer Science, vol. 60. Springer-Verlag, New York, 1978, pp. 393–481.
13. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
14. LAMPORT, L. The weak Byzantine generals problem. *J. ACM 30*, 3 (July 1983), 668–676.
15. LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.
16. PINTER, S. Distributed computation systems: Modelling, verification and algorithms. Ph.D. dissertation. Dept. of Computer Science, Boston Univ., Boston, Mass., 1984.
17. REISCHUK, R. A new solution for the Byzantine generals problem. *Inf. Control 64* (1985), 23–42.
18. SCHNEIDER, F. B. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst. 2*, 2 (May 1984), 145–154.
19. SKEEN, D. A quorum based commit protocol. Tech. Rep. TR 82-483, Computer Science Dept., Cornell Univ., Ithaca, N.Y., Feb. 1982.
20. SRIKANTH, T. K., AND TOUEG, S. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Rep. 84-623, Computer Science Dept., Cornell Univ., Ithaca, N.Y., 1984.

RECEIVED OCTOBER 1985; REVISED JULY 1986 AND MARCH 1987; ACCEPTED MARCH 1987