CONSENSUS IN THE PRESENCE OF PARTIAL SYNCHRONY

(Preliminary Version)

Cynthia Dwork
Laboratory for Computer Science
MIT
Cambridge, MA 02139

Nancy Lynch
Laboratory for Computer Science
MIT
Cambridge, MA 02139

Larry Stockmeyer
IBM Research Laboratory
San Jose, CA 95193

1. INTRODUCTION

1.1. Background

The problem of reaching agreement among separated processors is of fundamental importance to distributed computing, and has provided a rich set of interesting mathematical problems. (See [F] for a survey. Also see [GLPT,Sc,G,DLPSW,LM], for example.) One version of this problem considers a collection of N processors, p_1, ..., p_N, which communicate by sending messages to one another. Initially each processor pi has a value vi 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, e.g., crash (fail-stop faults), fail to send messages when they should (omission faults), or send erroneous messages (Byzantine faults).

Given assumptions about the properties of the message system and the processors and given the types of faults which can occur, one would like to know the maximum number of faults that can be tolerated; we call this number the *resiliency* of the system. For example, it might be assumed that there is a fixed bound Δ on the time for messages to be delivered (communication is synchronous), and a fixed 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 [LSP,DS] 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 $N > 3t$ [LSP,L1].

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, Dwork and Stockmeyer [DDS], building on earlier work of Fischer, Lynch and Paterson [FLP], prove that either if a fixed upper bound Δ on message delivery time does not exist (communication is asynchronous) or if 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 the consensus problem in practically motivated situations which lie between the completely synchronous and the completely asynchronous cases.

1.2. Partially Synchronous Communication

We first consider the case in which processors are synchronous (Φ exists and is known *a priori*) and communication lies "between" synchronous and

asynchronous. There are several 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 [FLP,DDS] 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 (we are assuming a lower bound on processor step time). 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 which 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 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 impossiblity results of [DDS] apply. The additional constraint is that there is a sufficiently large number L such that if at any time during the execution, the message system respects the upper bound Δ for L units of time, then all correct processors will reach a common decision sometime before the end of this "reliable interval". Moreover, the protocol never produces an inconsistent decision (two correct processors deciding differently) during the "unreliable period" when Δ does not hold.

The same argument as in the previous case shows one problem with treating lost or delayed messages in the same way as processor faults. There is also another problem with this idea. In typical systems, the loss or delay of a message is a much more likely event than a processor failure. Treating undesirable message behavior as processor faults tends to lead to a drastic overestimate of processor faults. Since consensus protocols introduce expensive mechanisms to cope with each additional processor fault, it seems better to separate consideration of the two kinds of events, and to try to use less costly mechanisms to cope with undesirable message behavior.

A third situation we consider is a technical variant on the second, which strengthens it in two ways. In this model, messages are never lost and Δ must hold from some point on after some finite "unreliable period". We prove that this model is equivalent to the first model, in which Δ exists but is unknown.

For succinctness, we say that communication is *partially synchronous* if one of these three situations holds: Δ exists but is not known *a priori*, or Δ is known but has to actually hold only for a sufficiently long period, or Δ is known and has to hold from some point on.

Our results determine precisely the maximum resiliency possible in cases where communication is partially synchronous, for four interesting fault models. For fail-stop or omission faults, we show that t-resilient consensus is possible iff $N > 2t$. For Byzantine faults with authentication, we show that t-resilient consensus is possible iff $N > 3t$. Also, for Byzantine faults without authentication, we show that t-resilient consensus is possible iff $N > 3t$. (The lower bound follows immediately from the result for the completely synchronous case in [LSP].) For the first three types of faults, the number of bits of communication required is a polynomial in N, t, and either (1) GST (the *global stabilization time*, or time when the messages start observing their required bound) for the models in which Δ holds eventually or sufficiently long, or (2) Δ for the model in which Δ is unknown. On the other hand, our algorithm for the unauthenticated Byzantine case uses an exponential amount of communication. We also have a t-resilient consensus protocol for Byzantine faults without authentication, which uses a polynomial amount of communication, but which requires $N > 4t$. (We do not know whether it is possible to obtain such a protocol for $3t < N \leq 4t$.)

Table 1 shows the maximum resiliency in various cases and compares our results with previous work. In each case, the table gives the smallest value of N for which there is a t-resilient protocol ($t \geq 1$). Except where indicated (by "exp") the algorithms require communication polynomial in N, t, and either GST or Δ.

It is interesting to note that for fail-stop, omission and Byzantine faults with authentication, the maximum resiliency for the partially synchronous case lies strictly between the maximum resiliency for the synchronous and asynchronous cases. It is also interesting to note that in the partially synchronous case, authentication does not improve resiliency. Results in the synchronous column are due to [LSP,DS,DFFLS], while those in the asynchronous column are due to [FLP,DDS].

Of the new results, the more interesting and difficult are the protocols and associated upper bounds. Our protocols use variations on a common method: a processor p tries to get other processors to change to some value v which p has found to be "acceptable"; p decides v if it receives sufficiently many acknowledgements 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. This general method and similar methods have already appeared in the literature, (cf. Skeen [Sk], Bracha and Toueg [BT]). Reischuk [R] and Pinter [P] have also obtained consensus results which treat message and processor faults separately.

1.3. Partially Synchronous Communication and Processors

It is easy to extend the models described in 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 for a sufficiently long period, or Φ can be known and actually have to hold from some point onward. We obtain results which completely characterize the resiliency in cases where both communication and processors are partially synchronous, for all four of the 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 during the same sufficiently long period, or both hold from some point on.

Surprisingly, the bounds we obtain are exactly the same as for the case where communication alone is partially synchronous. In the earlier case, the fact that Φ was known implied that each processor could maintain a local time that was guaranteed to be closely synchronized with the clocks 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 [L2].) 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 preceding case, the result is a new protocol for the new case.

1.4. Partially Synchronous Processors

In complete analogy to our treatment of partial communication synchrony, it is easy to define models where processors are partially synchronous and communication is completely synchronous (Δ exists and is known *a priori*). In Table 2 we summarize our results about N, the smallest number of processors for which t-resiliency is possible for each of the four fault models.

Technical Remarks:

Except where we have indicated otherwise, all of our protocols use only a polynomial amount of communication, that is, the number of bits of communication sent before all correct processors make a decision is polynomial in N, t, and either GST or Φ and Δ, depending on the particular model of partial synchrony.

Our protocols assume that an *atomic step* of a processor is to either receive a set of messages or send a message to a single processor, but not both; there is no 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 and broadcast to all processors in a single atomic step.

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.

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 results for the model in which processors are synchronous and communication is partially synchronous. The distributed clocks are defined in Section 4, where we also discuss how to combine our results of Section 3 with the clocks to produce protocols for the model in which both processors and communication are partially synchronous.

The results for the model in which communication is synchronous and processing is partially synchronous are omitted here for lack of space, as are the proofs of some of the results in Sections 3 and 4. All of the omitted material appears in [DLS], the complete version of the paper.

2. DEFINITIONS

2.1. Model of Computation

Our formal model of computation is based on the models of [FLP,DDS]. 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 which 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 them to p_i.

In the Send(m,p_j) instruction, p_j can be any processor, i.e., the communication network is completely connected. A processor's *state* is determined by the contents of its memory, including any special registers (e.g., program counter). 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 Receive instruction, the next state depends 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 steps. At each tick of real time, some processors each take one step of their protocols. A *run* of the system is described by specifying for each real-time step: (1) the processors which take steps, (2) the instruction which 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_i).

2.2. Failures

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

Fail-stop: The processor executes correctly but can stop at any time. Once stopped it cannot restart.

Omission: The processor executes correctly except that Send(m,p_j) might not place m in p_j's buffer.

Authenticated Byzantine: The processor exhibits arbitrary behavior. However, 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: The processor exhibits arbitrary behavior, and there is no mechanism for signatures. However, we assume that the receiver of a message knows the identity of the sender.

2.3. Partial Synchrony

Let $I = [t_1,t_2]$ be an interval of real time and let 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 steps, every correct processor takes 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.

(1) *delta is known*: there is a fixed Δ which holds in $[1,\infty)$ for every run R; this is the usual definition of *synchronous communication*.

(2) *delta is unknown*: for every run R there is a Δ which holds in $[1,\infty)$.

(3) *delta holds eventually*: there is a fixed Δ such that, for every run R, there is a time t0 such that Δ holds in $[t0,\infty)$, and no messages are lost in R.

(4) *delta holds sufficiently long*: there is a fixed Δ and sufficiently large L such that, for every run R, there is a time t0 such that Δ holds in [t0,t0 + L].

If (2), (3), or (4) hold, we say that communication is *partially synchronous*. In (3) and (4), t0 is called the *global stabilization time* (GST). In (4), L will in general depend on Δ, Φ and N. By replacing Δ by Φ above, (1) defines *synchronous processors*, and (2)-(4) define three types of *partially synchronous processors*.

Fix any of the four possible fault models. In [DLS] we show results that can be paraphrased as (4) --> (3), (3) --> (2) and (2) --> (3). Thus, in a sense, (2) and (3) are equivalent, in that the existence of a consensus protocol in one of these models implies the existence of a consensus protocol in the other, while (4) is a weaker model. However, this strengthens our results, since all our protocols work for the (4) variant, while all our lower bounds work for the (3) and (2) variants.

2.4. Correctness of a Consensus Protocol

Given assumptions A about processor and communication synchrony, given a fault mode F, and given 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 mode F, the protocol achieves:

Consistency. No two different processors in C decide differently.

Eventual Agreement. 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 then if any processor in C decides, then it decides v.

Weak Unanimity: if all initial values are v and C contains all processors, then if any processor decides, then it decides v.

3. PARTIALLY SYNCHRONOUS COMMUNICATION AND SYNCHRONOUS PROCESSORS

In this section we assume that processors are synchronous and communication is partially synchronous. Throughout most of this section we assume that the processor bound $\Phi = 1$ to simplify the exposition of the main ideas. Remarks at the end of the section then indicate several ways to extend the results to the case $\Phi > 1$. Since processors operate in lock-step synchrony, it is useful to imagine that each (correct) processor has a clock which 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.

The next three subsections give consensus protocols and lower bounds for the four types of faults.

3.1 Fail-Stop and Omission Faults

The consensus protocols in the following three subsections are all designed for the model in which Δ holds sufficiently long, and they handle arbitrary value domains V. In case $\Phi = 1$, as noted above, we can imagine that all (correct) processors have access to a common clock. Time, as measured by this clock, is divided into *phases*, and phases are subdivided into *rounds* of message exchange of length R each. The number $R = N + \Delta + 1$ is chosen large enough to allow processors to "broadcast" a message to all N processors (including themselves), and for all these messages to be received. Since our model does not have an atomic broadcast operation, this is done by sending the message to all processors, one at a time. Of course, our algorithms must allow for the possibility that a faulty processor could fail in the middle of a "broadcast", and for the possibility that messages sent before GST could be lost or arrive late. It will be seen that these possibilities do not affect the correctness of our algorithms. A processor always attaches a phase identifier (number) to messages, and any message sent during a phase h which arrives late during some phase $h' > h$ is ignored. Thus, one can imagine that communication during one phase is independent of communication during any other phase.

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 which 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.

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.

Processing is divided into alternating *trying* and *lock release* phases, with pairs of corresponding phases being numbered by consecutive integers starting with 1, where each trying phase is of length 3R and each lock release phase is of length R. We say that trying phase i mod N *belongs* to processor i.

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 mod N, it means that p thinks that processor i might decide v at phase k. Processor p only *releases* a lock if it learns that its supposition was false. A value v is *acceptable* to p if p does not have a lock on any value other than v.

We now describe the processing during a particular trying phase k. Let s denote the time of the beginning of the first round in phase k, and assume k $\equiv$ i mod N. At time s, each processor (including i) sends a list of all its acceptable values which are also in its PROPER set to processor i (in the form of a (list, k) message). (If V is very large or infinite, 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.) At time s + R, processor i attempts to choose a value to *propose*. In order for processor 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. It is possible that there might be more than one possible value which processor i might propose; in this case, processor i will choose one arbitrarily. Processor i then broadcasts a message (lock v,k).

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

If processor i receives acknowledgements from at least t + 1 processors by time s + 3R, then processor i decides v. After deciding v, processor i continues to participate in the algorithm.

Lock release phase k begins at time s + 3R. At time s + 3R, processors broadcast messages of the form (v,h), indicating that the sender 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 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 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 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: Immediately after any lock release phase which occurs completely in the interval [GST,GST + L] the set of values locked by processors contains at most one value.

Proof: Straightforward from the lock release rule. ▯

Theorem 4: Assume the model with fail-stop or omission faults, where the processors are synchronous with $\Phi = 1$ and communication is partially synchronous (Δ holds sufficiently long). Assume $N \geq 2t + 1$. Then Algorithm 1 achieves strong unanimity for an arbitrary value domain.

Proof:

First, we show that disagreement cannot be reached. Suppose that some correct processor i decides v at phase k, and this is the smallest numbered phase at which a decision is made. Then Lemma 2 imples that at all times after phase k, at least t + 1 processors have v locked. In consequence, 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 eventual agreement. Consider any trying phase, k, belonging to a correct processor, i, which is executed after a lock release phase, both occurring during [GST,GST + L]. We claim that processor i will reach a decision at trying phase k (if it has not done so already). By Lemma 3, there is at most one value locked by correct processors at the start of trying phase k. If there is such a 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. It follows that a proper, acceptable value will be found for processor i to propose, and that the proposed value will be decided upon by processor i at trying phase k. ∎

The following lower bound shows that the resiliency of Theorem 4 cannot be improved, even for weak unanimity and a binary value domain.

Theorem 5: Assume the model with fail-stop or omission faults, where the processors are synchronous and communication is partially synchronous (Δ holds eventually and no messages are lost). Assume $N \leq 2t$. Then there is no t-resilient consensus protocol which achieves weak unanimity for binary values.

Proof:

Assume the contrary, that there is an algorithm immune to fail-stop faults satisfying the required properties. We will 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 situation 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 after time t_A. The decision must be 0. For if it were 1, we could modify the situation to one where 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 situation, the processors in P still decide 1, contradicting weak unanimity.

Consider situation 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 after t_B steps for some finite t_B.

Consider situation 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 (resp., group Q) act exactly as they do in situation A (resp., situation B). This yields a contradiction. ∎

3.2. 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 set PROPER. (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 indeed 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.

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. As before, a value v is *acceptable* to p if p does not have a lock on any value other than v.

We now describe the processing during a particular trying phase k. Let s denote the time of the beginning of the first round in phase k, and assume $k \equiv i \bmod N$. At time s, each processor j (including i) sends a list of all its acceptable values which are also in its PROPER set to processor i, in the form E_j(list, k), where E_j is an authentication function. At time s + R, processor i attempts to choose a value to *propose*. In order for processor 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 which processor i might propose, then it will choose one arbitrarily. Processor i then broadcasts a message E_i(lock v,k,proof), where the proof consists of the set of signed messages E_j(list,k) received from the N - t processors which found v acceptable and proper.

If any processor receives a E_i(lock v,k,proof) message by time s + 2R, 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(lock v,k,proof) with the lock, and sends an acknowledgement to processor 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(lock v,k,proof) is said to be a *valid lock* on v at phase k.

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

Lock release phase k begins at time s + 3R. At time s + 3R, processors broadcast messages of the form E_i(lock v,h,proof), indicating that the sender has a lock on v with associated phase h and the given associated proof, and processor 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.

The proofs for Lemmas 6 through 8 and of Theorem 9 are analogous to the proofs of the corresponding results for Algorithm 1.

Lemma 6: It is impossible for two distinct values to acquire valid locks at the same trying phase, if that phase belongs to a correct processor. ∎

Lemma 7: 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. ∎

Lemma 8: Immediately after any lock release phase which occurs completely in the interval [GST,GST + L] the set of values locked by correct processors contains at most one value. ∎

Theorem 9: Assume the model with Byzantine faults and authentication where the processors are synchronous with $\Phi = 1$ and communication is partially synchronous (delta holds sufficiently long). Assume $N \geq 3t + 1$. Then Algorithm 2 achieves strong unanimity for an arbitrary value domain. ∎

The following lower bound result again applies in the case of weak unanimity and a binary value domain.

Theorem 10: Assume the model with Byzantine faults and authentication, where the processors are synchronous and communication is partially synchronous (delta holds eventually and no messages are lost). Assume $N \leq 3t$. Then there is no t-resilient consensus protocol which achieves weak unanimity for binary values. ∎

3.3. Byzantine Faults without Authentication

Here, we will describe two protocols. The first, simpler, protocol, is t-resilient and uses 4t + 1 processors. It uses a polynomial amount of communication. The second protocol needs only 3t + 1 processors, thereby achieving the maximum possible resiliency (as implied by the lower bound result of the previous section), but it uses more than a polynomial amount of communication.

Both algorithms are designed for the model in which Δ holds sufficiently long and for arbitrary value domains.

In both algorithms, the processors' PROPER sets are handled exactly as in Algorithm 2.

Algorithm 3: $N \geq 4t + 1$

Processing is again divided into alternating trying and lock release phases, with phases numbered as before. Now, however, the trying phases are of length 4R.

As before, at various times during the algorithm, processors may lock values. In algorithm 3, only a phase number is associated with every lock. As before, a value v is *acceptable* to p if p does not have a lock on any value other than v.

We now describe the processing during a particular trying phase k. Let s denote the time of the beginning of the first round in phase k, and assume $k \equiv i \bmod N$. At time s, each processor broadcasts a list of all its acceptable values which are also in its PROPER set, in the form (list, k). At time s + R, each processor p broadcasts a vector which says, for each processor q, which values q sent to p at the preceding round. At time s + 2R, processor i attempts to choose a value to *propose*. In order for processor i to propose v, it must have heard that each of at least N - 2t processors claims that at least N - 2t processors find value v acceptable and proper at phase k. As before, ambiguities are resolved arbitrarily. Processor i then broadcasts a message (lock v,k).

If any processor receives a (lock v,k) message by time s + 3R, and also has heard that each of at least N

3t processors claims that at least N - 2t processors find value v acceptable and proper at phase k, it locks v, associating the phase number k with the lock, and sends an acknowledgement to processor i. Release of other locks on v is handled as before.

If processor i receives acknowledgements from at least 3t + 1 processors, then processor i decides v. After deciding v, processor i continues to participate in the algorithm.

Lock release phase k begins at time s + 4R. At time s + 4R, processors broadcast messages of the form (v,h), indicating that the sender 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 t + 1 messages indicating that t + 1 distinct processors all have locks of the form (w,h') with $w \neq v$ and $h' \geq h$, then the processor releases its lock on v. (The values of w and h' need not be the same in all of these locks.)

Lemma 11: It is impossible for two distinct values to acquire locks by correct processors at the same trying phase, if that phase belongs to a correct processor.

Proof: The proof is similar to previous proofs and is left to the complete paper. ∎

Lemma 12: 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 2t + 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:

It is clear that at least 2t + 1 correct 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.

Then the lock is released during lock release phase l, when it is learned that at least t + 1 processors have locks on values $w \neq v$ with associated phases h, where $k \leq h \leq l$. Therefore, at least one correct processor, say j, has such a lock. Lemma 11 implies that no correct processor has a lock on any $w \neq v$ with associated phase k. Therefore, the correct processor j has a lock on $w \neq v$ with associated phase h, where $k < h \leq l$. In order for j to place this lock on w, at least N-3t processors each claim that at least N-2t processors find w acceptable at the first round of phase h. Since $N-3t \geq t+1$, at least one correct processor makes this claim, so at least N-2t processors actually find w acceptable. Since 2t + 1 correct processors have v locked at least through the first round of l, this is impossible. ∎

Lemma 13: Immediately after any lock release phase which occurs completely in the interval [GST,GST + L] either no value is locked or there exists some locked value v such that at most t correct processors hold locks on values other than v.

Proof: Straightforward from the lock release rule. (Consider some v whose lock is from the earliest phase from which any lock persists.) ∎

Theorem 14: Assume the model with Byzantine faults without authentication, where the processors are synchronous with $\Phi = 1$ and communication is partially synchronous (Δ holds sufficiently long). Assume $N \geq 4t + 1$. Then Algorithm 3 achieves strong unanimity for an arbitrary value domain.

Proof:

The proof that disagreement cannot be reached follows easily from Lemma 12 as in the proof of Algorithm 1.

Next, we argue eventual agreement. Consider any trying phase, k, belonging to a correct processor, i, which is executed after a lock release phase, both occurring during [GST,GST + L]. We claim that processor i will reach a decision at trying phase k (if it has not done so already). There are two cases. If some value v is locked at the beginning of trying phase k, then by Lemma 13, there is some locked value v such that at most t correct processors have values other than v locked at the start of trying phase k. Therefore, v is acceptable to at least $N-2t \geq 2t+1$ correct processors. Thus, by the beginning of trying phase k, these 2t + 1 correct processors have communicated to all correct processors that v is proper, so every correct processor will have v in its PROPER set. In the second case, no value is locked, so all values are acceptable. If there are at least t + 1 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 in either case that a proper, acceptable value will be found for processor i to propose.

Moreover, any value v which is proposed by processor i must have had N - 2t processors tell i that N - 2t processors found v to be acceptable and proper. Then at least N - 3t processors must tell all other processors that N - 2t processors found v to be acceptable and proper, so that all the correct processors will acknowledge the proposal. Thus, the proposed value will be decided upon by processor i at trying phase k. ∎

The second protocol of this section uses only $N \geq 3t + 1$ processors, but the amount of communication and the time to reach a decision after GST grows roughly like N^t in the worst case.

Algorithm 4: $N \geq 3t + 1$

Instead of rotating processors in successive phases, we rotate pairs (S,i), where S is a size N - t subset of the set of processors and i is a distinguished processor in that set. Each phase k is *owned* by the corresponding S, and the distinguished processor i plays the role of the coordinator.

Processing is again divided into alternating trying and lock release phases. We first describe the processing during a particular trying phase k. Assume that phase k is owned by the set S of N-t processors and that i is the distinguished processor. Each trying phase has four rounds. During the first round each processor in S broadcasts a list of all its acceptable values which are also in its PROPER set, in the form (list, k). Based on this information, processor i attempts to choose a value to propose. In order for processor i to propose v, it must have heard that all processors in S find v to be acceptable and proper. As before, ambiguities are resolved arbitrarily. During the second round, processor i broadcasts a message (propose v,k). If a processor j in S receives a message (propose v,k) from i and if j heard from all processors in S during the first round that v is acceptable and proper, then j broadcasts (lock v,k) during the third round. If a processor in S receives (lock v,k) messages from all in S, then it locks v and sends an acknowledgement to processor i. If processor i receives acknowledgemnts from all in S, then i decides v. After deciding, processor i continues to participate in the algorithm.

Each lock release phase has three rounds. During the first round, processors broadcast messages of the form (v,h) indicating that the sender has a lock on v at associated phase h. If a processor receives a message (v,h), then during the next two rounds it checks if (v,h) is *valid* by determining the set S of processors that owns phase h, and asking each processor in S whether it sent a message (lock v,h) at phase h. If at least N-2t processors in S respond affirmatively by the end of the third round then (v,h) is valid; otherwise it is not valid. If a processor has a lock on v with associated phase h and it receives a valid message (w,h') with $w \neq v$ and $h' \geq h$, then it releases the lock on v.

Lemma 15: 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:

It is clear that at least t + 1 correct processors lock v at phase k. Assume that the second conclusion is false. As before, 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.

Therefore, some correct processor received a valid message (w,h) during lock release phase l, where $w \neq v$ and $k \leq h \leq l$. Since (w,h) is valid, at least $N-2t \geq t+1$ processors said that they sent a message (lock w,h) at phase h. Therefore, at least one correct processor j actually sent (lock w,h). If h = k, then j would have sent both (lock w,k) and (lock v,k), which is impossible. Therefore, $k < h \leq l$. Since j sent (lock w,h), j heard during phase h that N-t processors (namely, the set that owns phase h) found w to be acceptable at phase h. But since at least t+1 correct processors have v locked at least through the first round of trying phase l, this is impossible. ∎

Lemma 16: Immediately after any lock release phase which occurs after GST, the set of values locked by correct processors contains at most one value.

Proof: Say that processor i has a lock on v with associated phase h and processor j has a lock on w with associated phase h' where $v \neq w$. Say that $h' \geq h$. During the lock release phase, i will receive the message (w,h') from j. Since j received the message (lock w,h') from at least N-t processors during trying phase h' and since at least N-2t of these are correct, i will determine that (w,h') is valid. Therefore i will release the lock on v. ∎

Theorem 17: Assume the model with Byzantine faults without authentication, where the processors are synchronous with $\Phi = 1$ and communication is partially synchronous (delta holds sufficiently long). Assume $N \geq 3t + 1$. Then Algorithm 4 achieves strong unanimity for an arbitrary value domain.

Proof:

The argument that disagreement cannot be reached is similar to before.

Next, we argue eventual agreement. Consider any trying phase, k, belonging to a set S consisting entirely of correct processors. Assume i is the distinguished processor at phase k. We claim that processor i will reach a decision at trying phase k (if it has not done so already). By Lemma 16, it follows as in previous proofs that a proper, acceptable value will be found for processor i to propose. Moreover, since all processors in S are correct, it is obvious that the entire trying phase k will complete successfully, and processor i will make a decision at the end. ∎

Our lower bound is tight for the case of unauthenticated Byzantine faults with no further restrictions. If we consider the problem with the requirement that communication be bounded by a polynomial, or that time be bounded by something linear in N after GST, then we do not know how to close the gap.

Remarks

1. Algorithms 1, 2 and 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 [H, FLa] show that t + 1 rounds are necessary even in case communication and processors are both synchronous and failures are fail-stop. A modification to all the algorithms is to have a processor broadcast the message "Decide v" whenever it decides v. This message is not tagged with a phase number, and other processors should accept a "Decide v" message at any time. 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, and these arguments are left to the reader. These modifications also give termination conditions for the processors, in models where no messages are lost. For fail-stop or omission faults, a processor can terminate after it broadcasts a "Decide v" message. For Byzantine faults, a processor can terminate after it has broadcast a "Decide v" message and has received "Decide v" messages from 2t other processors. In the model where messages can be lost before GST, it is not hard to argue that in any consensus protocol resilient to one fail-stop fault, at least one correct processor must continue sending messages forever. The argument is similar to Theorems 5 and 10.

2. We have described our algorithms for the model in which delta holds sufficiently long. We can then apply the model reductions mentioned at the beginning of section 2 to show that the same resiliency is possible in the model where delta is unknown. Although this is theoretically convenient, it may not give the most efficient protocols for the model where delta is unknown. An alternative is to modify the algorithms. Instead of using a fixed Δ to determine the length R of a round, Δ is increased as time progresses. For example, one might use $\Delta = 2^{\lfloor h/N \rfloor}$ during phase number h. If Δ' is the "actual" Δ that holds in the particular run that the the algorithm is executing, then the effective GST (the time when the increasing Δ reaches the actual Δ will be polynomial in N and Δ').

3. If $\Phi > 1$, we can again imagine that the processors have internal clocks, but that the clocks drift apart at a rate bounded above by Φ. One approach to designing a protocol for this model is to use one of the clock synchronization algorithms of [HSS, DHS, LL]. There are clock synchronization algorithms resilient to several Byzantine failures, even without authentication, and which have two properties: (1) the clocks of correct processors never differ by more than a fixed additive constant, and (2) the clocks of correct processors never run slower or faster than real time by more than some fixed multiplicative constant. Property (1) permits time to be divided into rounds so that no two correct processors are in different rounds at the same real time. Property (2) ensures that the algorithms run no slower than some constant times real time. Together, these two properties allow us to run the consensus protocols of this section, with processors reading their internal (private) clocks instead of a shared clock.

In Section 4 we show that the resiliency achieved by the protocols of this section can also be achieved if both processors and communication are partially synchronous. Of course, these stronger results imply that the same resiliency is achievable if communication is partially synchronous and processors are synchronous with some $\Phi > 1$, and this provides an alternate way of handling the case $\Phi > 1$.

4. PARTIALLY SYNCHRONOUS COMMUNICATION AND PROCESSORS

In this section we show that the protocols of the previous section can be modified to work, with the same resiliencies, in models where both communication and processors are partially synchronous. Moreover, algorithms 1, 2, and 3 will still use a polynomial amount of communication. We describe the modified protocols in detail for the case where Φ and Δ both hold sufficiently long; that is, there are fixed constants Φ and Δ, such that for any run, there is a time GST such that both Φ and Δ hold in the interval [GST,GST + L] for a sufficiently large L depending polynomially on N, Φ, and Δ. As described in Remark 2 at the end of the previous section, the protocols can be modified for the model where both Φ and Δ are unknown by letting the "built in" Φ and Δ increase as time progresses.

In the previous section, the processors had a common notion of time which allowed time to be divided into phases. If Φ does not always hold, no such common notion of time is available. Therefore, the first step is to describe a protocol which gives the processors some approximately common notion of time, at least during the reliable interval [GST,GST + L]. We call such a protocol a *distributed clock*. Each processor has a private (software) clock. Before GST, the private clocks of correct processors could be very far apart. However, during the reliable interval [GST,GST + L] there are two correctness conditions which the private clocks of all correct processors must satisfy: within some constant amount of real time after GST (1) the private clocks must grow at a rate within some constant factor of real time, and (2) at any real time the difference in the values of any two private clocks is bounded above by an additive constant known to the processors. The three "constants" here depend polynomially on N, Φ and Λ.

Once we have defined the distributed clocks, the protocols of the previous section are modified by letting each processor use its private clock to determine which round (and therefore, which phase) it is in. For convenience, processors alternate receiving and sending operations. Alternate pairs of receive-send operations are used to maintain the distributed clock, with the other receive-send pairs being used by the consensus protocol. We first describe what happens during the clock maintenance steps for two different distributed clocks. The first handles Byzantine faults without authentication and requires $N \geq 3t + 1$. The second handles Byzantine faults with authentication and requires $N \geq 2t + 1$. (In [DLS] we define another distributed clock which handles only fail-stop faults, but is N-resilient. This clock is not needed for the results presented in this paper.)

4.1. A Distributed Clock for Byzantine Faults without Authentication

Throughout this section we assume that $N \geq 3t + 1$. The term *step* refers to a real-time step; real-time steps are numbered 0, 1, 2 ... . Processors participate in our distributed clock protocols by sending *ticks* to one another. For convenience, we define a *master clock* whose value at any step 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". A $\geq$*i-tick* is a j-tick for any $j \geq i$. We say p has *broadcast an i-tick* if it has sent a $\geq$i-tick to all N processors.

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

We adopt the convention that all processors have exchanged ticks and claims of size 0 before step 0. These messages are not actually sent, but they are considered to have been sent and received.

The *master clock*, $C: N \rightarrow N$, is defined at any real-time step s by

C(s) = maximum j such that t + 1 correct processors have broadcast a j-tick by the beginning of step s.

Since all processors are assumed by convention to have broadcast a 0-tick before step 0, C(0) = 0.

For each processor p_i the *private clock*, $c_i: N \rightarrow N$, is defined by

$c_i(s)$ = maximum j such that at the beginning of step s p_i has received either (1) $2t + 1$ $\geq$j-claims or (2) messages from t + 1 processors, where each message is either a $\geq(j + 1)$-tick or a $\geq(j + 1)$-claim.

Since p_i is assumed to have received 0-claims from all N processors by step 0, $c_i(0) = 0$.

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 step 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 step s. We will show in Lemma 18 that $c_i(s) \leq C(s)$, so p_i will never force the master clock to skip a value. We will also show that "soon" after GST the value of the master clock exceeds those of the private clocks by only a constant amount, so during the reliable interval 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, ticks and claims. The private clock of p_i is stored in a local variable ci. Processor p_i updates its private clock every time it executes a receiving clock maintenance operation 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 fourth step that p_i takes). The following two programs describe the tick and claim broadcasting procedures. A processor begins the distributed clock protocol by setting ci to 0 and calling TICK(0), where TICK(b) is the protocol shown in Figure 1. Note that the value of ci may change during an execution of TICK(b), but the claim is made only for a (b + 1)-tick. This is consistent with our definition of what it means to have broadcast a (b + 1)-tick.

```
TICK(b):
j <-- 0;
while j < N do
begin
  j <-- j + 1;
  send (ci + 1)-tick to p_j;
end;
call CLAIM(b).

CLAIM(b):
send (b + 1)-claim to all processors;
if ci > b then call TICK(ci); else call CLAIM(b).
```

Figure 1: Procedure TICK

The proofs of Lemmas 18 and 19 are fairly straightforward from the definitions and the protocol. Lemma 20 is proved by a simple induction, using Lemma 19.

Lemma 18: For all $s \geq 0$ and for all i such that p_i is correct, $c_i(s) \leq C(s)$. ∎

Lemma 19: For all $s \geq 0$ the largest tick sent by a correct processor at step s has size at most C(s) + 1. ∎

Lemma 20: For all $s,x \geq 0$, $C(s + x) \leq C(s) + x$. ∎

The above lemmas are independent of both communication and processor synchrony.

The next few lemmas discuss the behavior of the clocks during the reliable interval I = [GST,GST + L]. Lemma 21 says that the private clocks increase at most a constant factor more slowly than real time. Lemma 23 has two parts. The first says that the master clock exceeds the value of the private clocks by at most an additive constant. This, together with Lemma 18, bounds the difference between any two private clocks at any instant of real time. The second part of Lemma 23 says that, at least during the reliable interval, the master clock runs at a rate at most a a constant factor more slowly than real time. Let $D = \Delta + 4\Phi$. Note that if a message is sent to a correct processor p at step s $\geq$ GST and s + D is in I, then p will receive the message by step s + D; the message will be delivered by step $s + \Delta$ and within 4Φ more steps p will execute a receiving clock maintenance operation.

Lemma 21: Let s and j be such that $s \geq$ GST, $s + 16N\Phi + D$ is in I, and $c_i(s) \geq j$ for all correct p_i. Then $c_i(s + 16N\Phi + D) \geq j + 1$ for all correct p_i. ∎

Lemma 22: Let T = C(GST). Then $C(GST + 52N\Phi + 4D) \geq T + 2$. ∎

Lemma 23: Let s_0 be the minimum s such that C(s) = C(GST) + 2 (s_0 exists by Lemma 22).

(1) For all x in I such that $x \geq s_0 + D$ and for all correct processors i, $c_i(x) \geq C(x)\text{-}D\text{-}1$.

(2) For all $y \geq s_0$ such that $y + 32N\Phi + 3D$ is in I $C(v + 32N\Phi + 3D) \geq C(y) + 1$. ∎

Lemmas 18, 20, 21, and 23 yield the correctness conditions which must be satisfied by the private clocks of all correct processors. Specifically, Lemma 21 says that the private clocks do not grow too slowly, while Lemmas 18 and 20 say they do not grow too quickly. That is, within a constant amount of time after GST the private clocks grow within a constant factor of real time. As pointed out above, Lemmas 18 and 23 (1) say that soon after GST the private clocks of any two correct processors differ by at most a known, additive constant, at least during the reliable interval.

4.2. 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". A $\geq$*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 $t + 1$ $\geq$(i-1)-claims each signed by a different processor.

A $\geq$*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: N → N is defined by

C(s) = maximum j such that some correct processor has broadcast a j-tick by the beginning of step s.

The *private clock* c_i: N → N is defined by

$c_i(s)$ = maximum j such that p_i has received $t + 1$ $\geq$j-claims (from different sources), either directly, or

indirectly as part of a tick, by the beginning of step 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 which it can construct (this will always be a $\geq$(b+1)-tick). A correct processor will ignore any received j-claim if it does not come with an attached $\geq$j-tick.

Lemma 24: Lemmas 18, 19, 20, 21, 22, and 23 hold for the authenticated Byzantine clock. ∎

In addition, we need one more lemma to support our claim of a polynomial amount of communication. The proof is immediate from the definitions.

Lemma 25: Any tick or claim sent by a correct processor at step s can be represented by O(t log(C(s))) bits. ∎

4.3. Using the Clocks

As described above, alternate pairs of receive-send operations are used to maintain a distributed clock, and the other receive-send pairs are used to run one of the protocols of Section 3. For Algorithm 1 (fail-stop and omission faults) we use the authenticated Byzantine clock, simplified appropriately because the signatures are not needed and because we cannot assume the authentication capablity. Note that the consensus protocol and the distributed clock protocol have the same constraint on the number of processors, $N \geq 2t+1$. For Algorithms 3 and 4 (unauthenticated Byzantine faults), we use the unauthenticated Byzantine clock. For Algorithm 2 (authenticated Byzantine faults) either clock could be used. For all four algorithms L, the length of the reliable period, is somewhat larger in the new model.

Processing is divided into alternating *rounds* and *waiting periods* of length R and W respectively. Specifically, $R = 4N\Phi + \Delta + 4\Phi$ is the time required for N processors to broadcast a message and for this message to be received, and $W = 52N\Phi + 4(\Delta + 4\Phi)$ is the maximum difference between the private clocks of any two correct processors during $[GST + s_1, GST + L]$, where $s_1 = 52N\Phi + 5(\Delta + 4\Phi)$ (see Lemmas 18, 22, and 23). When running the consensus protocol, a processor uses its private clock to determine its current phase and round. In addition to labelling messages with phase numbers, processors label messages with round numbers. During any given round, only messages labelled with the same round number are accepted: other messages are ignored. During any given waiting period, only messages from either of the two adjacent rounds are accepted. No messages are sent during waiting periods.

For all four of the consensus protocols, the proofs that no two correct processors decide differently are identical to the proofs given in Section 3, since at no point in those proofs did we use the fact that different processors are executing the same phase at the same real time. For example, in Algorithm 1, if a processor i decides v at its phase k, then at least t+1 processors lock v at their phase k, and one argues as in Lemma 2 that these locks will never be released at any higher numbered phase.

To argue eventual agreement after GST, note that by choice of W no two correct processors are simultaneously executing different rounds at the same time x, for any x in the interval $[GST + s_1, GST + L]$. Further, any message labelled with a given round, say k, and sent to a correct processor during $[GST + s_1, GST + L-D]$, will be received and accepted before that processor begins round k+1. We now choose the lengths T_T and T_R of phases large enough so that all required communication during a phase will have time to complete, at least for all phases which take place entirely within $[GST + s_1, GST + L]$.

Theorem 26: Assume the model where communication and processors are both partially synchronous (delta and phi both hold sufficiently long). If Algorithms 1, 2, 3 and 4 are modified as decribed above, Theorems 4, 9, 14, 17 still hold. ∎

Our claims that the modified algorithms 1, 2 and 3 use a polynomial amount of communication and that agreement is reached within a polynomial amount of real time after GST follow from the fact that the master clock, during $[GST + s_1, GST + L]$, is running at a rate no slower than $1/(32N^*\Phi + 3(\Delta + 4\Phi))$ times real time (see Lemma 23).

The results for the case in which processors are partially synchronous and communication is synchronous are deferred to the complete paper.

Acknowledgment. Joe Halpern asked whether the impossibility results of [FLP,DDS] 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 Φ or Δ are unknown.

REFERENCES

[BT] Bracha, G., and Toueg, S., "Resilient consensus protocols," Proc. 2nd PODC, 1983, pp. 12-26.

[D] Dolev, D., personal communication.

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

[DFFLS] Dolev, D., Fischer, M.J., Fowler, R., Lynch, N.A., and Strong, H.R., "Efficient Byzantine agreement without authentication," Information and Control (to appear); see also IBM Research Report RJ3428 (1982).

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

[DLPSW] Dolev, D., Lynch, N., Pinter, S., Stark, E. and Weihl, W., "Reaching approximate agreement in the presence of faults," Proceedings of 3rd Annual IEEE Symposium on Reliability in Distributed Software and Database Systems,1983.

[DLS] Dwork, C., Lynch, N., Stockmeyer, L., "Consensus in the Presence of Partial Synchrony," in preparation.

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

[F] Fischer, M., "The consensus problem in unreliable distributed systems (a brief survey)," Report YALEU/DCS/RR-273, Yale University, June, 1983.

[FLa] Fischer, M. and Lamport, L., "Byzantine Generals and Transaction Commit Protocols," SRI Technical Report, Op. 62.

[FL] Fischer, M. and Lynch, N., "A lower bound for the time to assure interactive consistency," Info. Proc. Lett. 14, 4 (1982), pp. 183-186.

[FLP] Fischer, M., Lynch, N. A. and Paterson, M., "Impossibility of distributed consensus with one faulty process," Proc. of 2nd Symposium on Principles of Database Systems, Atlanta, GA, 1983.

[G] Garcia-Molina, H., Pittelli, F., and Davidson, S., "Is Byzantine agreement useful in a distributed database?" Manuscript.

[GLPT] Gray, J.N., Lorie, R.A., Putzulo, G.R., and Traiger, I.L., "Notes on Database Operating Systems," Operating Systems: An Advanced Course, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, N.Y., 1978, pp. 393-481.

[H] Hadzilacos V., "A Lower Bound for Byzantine Agreement with Fail-stop Processors," TR-21-83, Harvard University, 1983.

[HSS] Halpern J., Simons, B., and Strong, H. R., "An efficient fault-tolerant algorithm for clock synchronization," Report RJ 4094, IBM Research Division, San Jose, CA, Nov., 1983.

[L1] Lamport, L., "The weak Byzantine generals problem," J.ACM 30 (1983), pp. 668-676.

[L2] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," CACM 21, No. 7, (1978), pp. 558-564.

[LL] Lundelius, J., and Lynch, N. "A New Fault-Tolerant Algorithm for Clock Synchronization" These Proceedings.

[LM] Lamport, L., and Melliar-Smith, P.M., "Synchronizing clocks in the presence of faults," Technical Report, SRI International, March, 1982.

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

[P] Pinter, S., "Distributed computation systems: modelling, verification and algorithms," PhD Thesis, Boston University, 1984.

[R] Reischuk, R., "A new solution for the Byzantine generals problem," IBM Report RJ3673, November, 1982.

[Sc] Schneider, F.B., "Byzantine generals in action: implementing fail-stop processors," Manuscript, August, 1983.

[Sk] Skeen, D., "A quorum based commit protocol," Report TR 82-483, Dept. of Computer Science, Cornell Univ., Feb., 1982.

Failure mode	Synchronous	Asynchronous	Partially Synchronous
Fail-stop	t	∞	2t+1
Omission	t	∞	2t+1
Byzantine with Authentication	t	∞	3t+1
Byzantine without Authentication	3t+1	∞	3t+1 (exp) $3t \leq N \leq 4t+1$

Table 1: Smallest number of processors N ($N \geq 2$) for which there exists a t-resilient consensus protocol ($t \geq 1$).

Fail-stop:	$N = t$
Omission:	$t \leq N \leq 2t + 1$
Byzantine with Authentication:	$N = 3t + 1$ $2t \leq N \leq 2t + 1$ for the case of "weak unanimity" [F]
Byzantine without Authentication:	$N = 3t + 1$ (exponential communication) $3t < N \leq 4t + 1$ (polynomial communication)

Table 2: The smallest number of processors N for which t-resiliency is possible in the model with synchronous communication and partially synchronous processors.