Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults*

(Preliminary Version)

Shlomi Dolev† Jennifer L. Welch‡

Abstract

We initiate a study of bounded clock synchronization under a more severe fault model than that proposed by Lamport and Melliar-Smith [LM-85]. Realistic aspects of the problem of synchronizing clocks in the presence of faults are considered. One aspect is that clock synchronization is an on-going task, thus the assumption that in *any* period of the execution at least two thirds of the processors are nonfaulty is too optimistic. To cope with this reality we suggest *self-stabilizing* protocols that stabilize in any (long enough) period in which less than a third of the processors are faulty. Another aspect is that the clock value is bounded. A single transient fault may cause the clock to reach the upper bound. Therefore we suggest a bounded clock that wraps around when appropriate.

We present two randomized self-stabilizing protocols for synchronizing bounded clocks in the presence of Byzantine processor failures. The first protocol assumes that processors have a common pulse, while the second protocol does not. A new type of distributed counter based on the Chinese remainder theorem is used as part of the first protocol.

1 Introduction

In a distributed system, it is often necessary to keep the logical clocks of the processors synchronized. In such a system physical clocks may drift and messages could have varying delivery times. Moreover, processors may be faulty, and in many cases the type of failures is not predictable in advance. To handle this situation, the worst type of failures must be considered, namely *Byzantine* faults [LSP-82]. In the presence of Byzantine faults a processor can exhibit arbitrary "malicious", "two faced", behavior.

The problem of keeping clocks synchronized in the presence of Byzantine faults has been extensively studied (e.g., [HS+-84, LM-85, MS-85, DHS-86, ST-87, WL-88, RSB-90]). Lamport and Melliar-Smith [LM-85] were the first to present the problem and show that $3f+1$ processors are sufficient to tolerate f Byzantine faults. The necessity of $3f+1$ processors to tolerate f faults was later proved in [DHS-86]. A weaker fault model called authenticated Byzantine allows a protocol that can tolerate any number of faulty processors [HS+-84]. In that failure model reintegration of repaired processors is only possible if less than half the processors are faulty. Many of the protocols for this problem assume that the clocks are initially synchronized and thus focus on keeping them synchronized in the presence of clock drift.

The problem of how to ensure that the clocks are initially synchronized was addressed in, e.g., [ST-87, WL-88]. In these protocols, some mechanism is assumed that allows all the nonfaulty processors to begin the protocol within a bounded time period of each other. The mechanism essentially is that the processes

*Supported in part by TAMU Engineering Excellence funds and NSF Presidential Young Investigator Award CCR-9158478.

†Department of Mathematics and Computer Science, Ben-Gurion University, Beer-Sheva, 84105, Israel. e-mail: shlomi@cs.bgu.ac.il.

‡Department of Computer Science, Texas A&M University, College Station, TX 77843. e-mail: welch@cs.tamu.edu.

are assumed to wake up in a distinguished initial state, in which they can uniquely perform initializing actions, including communication with each other.

In this work we weaken the assumptions made for the design of clock synchronization protocols in the presence of Byzantine faults. Our goal is protocols that cope with a more severe (and realistic) fault model than the traditional Byzantine fault model [LSP-82]. Initially, protocols that tolerate Byzantine faults were designed for flight devices that need to be extremely robust. In such a device the traditional assumptions could be violated: Is it reasonable to assume that during *any* period of the execution less than one third of the processors are faulty? What happens if for a short period more than a third are faulty (perhaps experience a weaker fault than a Byzantine fault)? What happens if messages sent by nonfaulty processors are lost in one instant of time?

In this paper we present *self-stabilizing* protocols that can overcome these problems. Such temporary violations of the assumptions can be viewed as leaving the system in an arbitrary initial state from which the protocol resumes. Self-stabilizing protocols work correctly when started in any initial system state. Thus, even if the system loses its consistency due to an unexpected temporary violation of the assumptions made (e.g., more than one-third faulty, unexpected message loss) the system synchronizes the clocks when subsequently the assumptions hold (e.g., less than a third experience Byzantine faults).

Originally, Dijkstra defined (in [Dij-74]) a protocol to be *self-stabilizing* if, when started in an arbitrary system state, the system converges to a consistent global state that realizes the task. Self-stabilizing protocols are resilient to *transient faults* – faults that cause the state of a processor to change arbitrarily and then from the new state, the processor resumes operation according to its program. A *permanent fault* is a fault that causes a processor to permanently misbehave. A protocol tolerates *hybrid faults* if it is resilient to both transient and permanent faults (e.g., [DW-93, GP-93] which consider *napping* and *omission* faults, respectively). We are interested in clock synchronization protocols that can tolerate hybrid faults: they should work from an arbitrary initial configuration and they should tolerate less than a third of the processors exhibiting permanent Byzantine faults.

A realistic assumption for a clock synchronization protocol is that a 64-bit clock is "unbounded" for most possible applications. However, in the context of self-stabilizing protocols transient faults could cause the system to reach the upper bound of the clock at once. Thus, another aspect of the problem should be considered: the fact that the clocks are bounded.

In this paper we present two randomized self-stabilizing clock synchronization protocols that work in the presence of Byzantine faults. Both protocols work for bounded clocks. The first assumes the existence of a common pulse while the second does not make this assumption. The expected stabilization time of both protocols is exponential in n. This is a drawback when the number of processors is large. However, in addition to being of theoretical interest, we believe that our protocols could be of practical interest, at least when the number of backup processors is small.

One of the contributions of this paper is an interesting usage of the Chinese remainder theorem for implementing a distributed counter. This counter is used to accelerate the first protocol.

The remainder of the paper is organized as follows. In the next section we formalize the assumptions and requirements for the protocol. Section 3 presents a clock synchronization protocol under the assumption of a common pulse. In Section 4 we present a protocol that does not assume the existence of common pulses. Conclusions are in Section 5.

2 Definitions

A distributed system consists of a set of processors that communicate by sending messages to each other. Messages have a bounded delay. Each processor has a bounded physical clock that is constantly incremented, wrapping around when appropriate; the physical clocks at the different processors run at approximately the same rates. Each processor also has a bounded logical clock, which is computed as a function of the current state and physical clock value. The goal is for the logical clocks of the nonfaulty processors to become and subsequently remain close to each other, while continuing to progress at a reasonable rate (wrapping around when appropriate). We consider two types of timing behavior of the system, synchronous and semi-synchronous. In both models, processors take steps either when they receive a message, or when their physical clocks reach some predetermined value. In addition, in the synchronous model, there is a common pulse that periodically occurs simultaneously at all processors, causing them to take a step. We now proceed more formally.

Each *processor* P_i, $1 \leq i \leq n$, is modeled as a state machine. Associated with the processor is its *physical clock*, which takes on integral values from 0 to $M_{pc} - 1$ for some M_{pc}. The state contains a distinguished *timer* variable that can take on the values 0 to $M_{pc} - 1$ and nil; it indicates that the processor wants to take a step the next time its physical clock has the given value. A transition takes the current state of the processor, the current value of its physical clock, and a message received (if any) and produces a new state of the processor and a set of messages to be sent. The *message system* holds all messages sent but not yet received. A *configuration* of the system is a set of processor states, one per processor, a set of physical clock values, one per processor, and a state for the message system.

An execution is an alternating sequence of configurations and events $C_0, e_1, C_1, \cdots$. In a *semi-synchronous execution*, events happen at real times, taking one configuration to the next. There are two types of events. One type is a *tick* of some processor's physical clock, causing it to increase by 1 mod M_{pc}. Nothing else changes. We require that the real time elapsed between two successive ticks of the same processor be between $1 - \rho$ and $1 + \rho$ for some fixed ρ.

The other type of event is a *step* of some processor. No processor can take more than one step at the same real time. In the step, the processor may or may not receive a message. The real time elapsed between the sending and receiving of any message must be in the range $[d - \epsilon, d + \epsilon]$ for some fixed d and ϵ. There is a fixed set of *faulty* processors of size f, where $n > 3f$. If the processor taking the step is nonfaulty, then the succeeding configuration must correctly reflect the processor's transition function acting on the message received and the state and physical clock in the preceding configuration. Thus the only changes are to the processor's state and the message system (removing the message received and adding the messages sent). If the processor taking the step is faulty, it can change state arbitrarily and add arbitrary messages (from itself) to the message system.

In a *synchronous execution*, in addition to the above constraints, there exists a value $\pi > 0$ such that, for all i, every processor P_i receives a special Pulse message (from a dummy processor) at time $i \cdot \pi$. (I.e., all the processors take a step at each pulse and the pulses occur regularly with period π.)

We require that for every processor P_i there exist a function $clock_i$ that, given a state of P_i and a value for P_i's physical clock, returns a value in the range 0 to $M_{lc} - 1$ for some fixed M_{lc}. This is the logical clock of P_i. Given a particular execution $C_0, e_1, \ldots$, we denote by $clock_i(t)$ the value of the function $clock_i$ applied to P_i's state and physical clock value in C_j, where j is the configuration in the execution whose real time of occurrence is the largest not exceeding t. We require that there exist a finite time t_s for which the following two conditions hold:

Clock Agreement: There exists $\gamma < M_{lc}/4$ such that for all $t \geq t_s$ and all nonfaulty processors P_i and P_j: $clock_i(t) - clock_j(t) \pmod{M_{lc}} \leq \gamma$.[1]

Clock Validity: There exists Δ, $0 < \Delta \leq M_{lc}/4$, and there exists $a \geq 0$ such that for all real times $t > t_s$ and all i, if $clock_i(t) = T$, then $T + \Delta/(1+a) \bmod (M_{lc}) \leq clock_i(t + \Delta) \leq T + (1+a)\Delta \bmod (M_{lc})$.

[1]The constant 4 is chosen for convenience; any constant larger than 2 is sufficient. Note that if the constant is 2 then this condition holds for any arbitrary configuration, since every two clock values are at most $M_{lc}/2$ apart.

Clock Agreement states that after t_s, the difference between any two nonfaulty processors' clocks is at most γ. Clock Validity states that after t_s, the amount of logical clock time that elapses during Δ real time is a linear function of Δ.

3 Synchronous Protocol

We first describe a protocol for the synchronous system, in which nonfaulty processors have access to a periodic common pulse. Each pulse triggers the processors to synchronize their clocks. The time between two successive pulses appears to be an important parameter to the problem. In case two successive pulses are farther apart than the time required to run a Byzantine agreement protocol, then the following scheme solves the problem: Every pulse starts a new version of the Byzantine agreement to agree on the common clock value. However, when the pulses are only on the order of the round trip message delay apart, this scheme cannot work.

We assume that the pulses are on the order of the round trip delay apart. Recall that π is the time between two successive pulses. Nonfaulty processors send messages and update their logical clocks only when a pulse occurs. We assume that π is long enough such that when a pulse takes place, no message sent by a nonfaulty processor in the previous pulse is present in the system. Whenever a nonfaulty processor P is triggered by a pulse, P sends a message with its clock value to all its neighbors. Then P waits to receive all the clock values of the other processors. P waits for a period $(1+\rho)(d+\epsilon)$ that is longer than the bound on the message delay and accounts for clock drift. If during that period P receives more than one message from some neighbor, say Q, then P uses the latest value that arrives from Q. Thus, at the end of such a period P has a set of at least $n-f$ logical clock values, at most one value for each nonfaulty processor including P. P uses the set of the logical clocks received in order to choose its own clock value.

The formal description of the protocol appears in Figure 1. We now describe the protocol informally. The protocol for a processor P works as follows: (1) if the value of P's clock appears less than $n-f$ times in the set of the received logical clocks then P assigns 0 to its clock. Otherwise, (2) in case that the value of P's clock appears at least $n-f$ times, we further distinguish between the case (2.1) in which P's clock value is not equal to 0 and the case (2.2) in which it is equal to 0. In case (2.1) P increments its clock by 1 (modulo the number of clock values M_{lc}). Case (2.2) is further subdivided into two cases: (2.2.1) in which (according to the state of P) in the previous pulse P incremented its clock by 1 (and the result was 0) and the case (2.2.2), otherwise. In case (2.2.1) P increments its clock by 1 (to be 1). In case (2.2.2) P tosses a coin and assigns the result (0 or 1) to its clock.

The protocol guarantees (with probability 1) that the system eventually reaches a global state in which all the nonfaulty processors have the clock value 1. Once such a global state is reached the clocks are synchronized: In every pulse, every nonfaulty processor P receives messages from at least $n-f-1$ processors containing a clock value that is identical to its own clock value. Moreover, a pulse in which all the nonfaulty processors set their clocks to 0 always follows a pulse in which every nonfaulty processor *increments* its clock value by 1 to set it to 0. Thus, case (2.2.2) is not applied.

The main idea of the protocol is to ensure that only when there are "enough" nonfaulty processors with the same clock value will this value be incremented. It is proved in the sequel that in any pulse at most one clock value of nonfaulty processors is incremented by 1 while the rest of the values are changed to be zero. This ensures that after the first pulse, the set of clock values of the nonfaulty processors contains at most two elements. Moreover, if two such elements indeed exist one of them is 0.

At first glance this seems to be sufficient and no coin toss is needed; the value that is incremented will eventually wrap around to 0 and at that time the clocks of all the nonfaulty processors will be 0. However, we now describe an infinite execution, E, that does not use coin tosses in which the clocks never become synchronized. Consider a system with four processors P_1, P_2, P_3 and P_4 in which P_4 exhibits Byzantine behavior. Let $0, 0, 1$ be the clock values of P_1, P_2, P_3, respectively, in the first configuration of E. In the first pulse P_4 sends clock value 1 to P_1 and P_3 and clock value 0 to P_2. Thus, P_1 receives the clock values vector $0, 0, 1, 1$, P_2 receives $0, 0, 1, 0$ and P_3 $0, 0, 1, 1$. P_2 is the only processor that finds $n - f = 3$ processors with the same clock value (namely, the clock value 0) and increments its clock value by one (to be 1). At the same time, P_1 and P_3 find two clock values with value 1 and two with value 0 and assign 0 to their clocks. Hence, a configuration with clock values $0, 1, 0$ for P_1, P_2, P_3, respectively, is obtained. P_4 continue and sends the clock values $1, 1, 0$ to P_1, P_2, P_3, respectively. P_1 receives the clock values vector $0, 1, 0, 1$, P_2 receives $0, 1, 0, 1$ and P_3 receives $0, 1, 0, 0$. Similarly, P_3 is the only processor that finds $n - f = 3$ processors with the same clock value and assigns 1 to its clock while P_1 and P_2 assign 0. We reach a configuration with clock values $0, 0, 1$ for P_1, P_2, P_3 which are identical to the clock values in the first configuration. Therefore, an infinite execution in which nonfaulty processors never agree on their clock values is possible.

To overcome the above problem we use coin tosses. In a pulse in which a nonfaulty processor with clock value 0 receives $n - f$ clock values with value 0 the processor tosses a coin and decides whether to assign 0 or 1 to its clock. This leads to a possible scenario (that has some probability of occurring) in which the coin toss results cause all the nonfaulty processors to simultaneously assign 1 to their clocks.

```
when pulse occurs:
  broadcast clock_i
  collect clock values until (1 + ρ)(d + ε) time has elapsed on the physical clock
  if |{j|clock_i = clock_j}| < n − f then (*case (1)*)
     {clock_i := 0; last_increment_i := false}
  else (*case (2)*)
       if clock_i ≠ 0 then (*case (2.1)*)
          {clock_i := (clock_i + 1) mod M_lc; last_increment_i := true}
       else (*case (2.2)*)
            if last_increment_i = true then (*case (2.2.1)*) clock_i := 1
            else (*case 2.2.2*) clock_i := toss(0, 1)
            if clock_i = 1 then last_increment_i := true
            else last_increment_i := false
```

Figure 1: The Synchronous Protocol for P_i

3.1 Correctness Proof of the Synchronous Protocol

Throughout the proof we say that a processor P_i *increments* its clock by 1 in a certain pulse, if P_i assigns $last\_increment := true$ during this pulse. Otherwise, we say that P_i *assigns* 0 to $clock_i$.

Lemma 3.1 *If nonfaulty processors P_i and P_j increment their clocks by 1 during some pulse $\mathcal{P}$, then immediately after $\mathcal{P}$, $clock_i = clock_j$.*

Proof: Assume towards contradiction that $clock_i = (x+1) \bmod M_{lc} \neq clock_j = (y+1) \bmod M_{lc}$ following $\mathcal{P}$. Hence, during $\mathcal{P}$, P_i finds at least $n - f$ clock values that are equal to x. At least $n - 2f$ of them belong to nonfaulty processors. Thus, P_j also receives $n - 2f$ clock values that are equal to x. Hence, P_j receives at most $n - (n - 2f) = 2f$ clock values that are equal to y. Since $n > 3f$, it holds that $n - f > 2f$, which contradicts the possibility of P_j receiving at least $n - f$ clock values that are equal to y. ■

Lemma 3.1 implies in a straightforward manner the correctness of the next two corollaries.

Corollary 3.2 *After every pulse, the set of clock values of the nonfaulty processors contains at most two elements. In case there are such two values, one of them is* 0.

Corollary 3.3 *If during a pulse* $\mathcal{P}$ *a nonfaulty processor* P *increments its clock value by* 1 *and the result is* 0, *then immediately following* $\mathcal{P}$ *the clock values of all the nonfaulty processors are* 0.

Claim 3.4 *If during a pulse* $\mathcal{P}$ *that follows the first pulse, a nonfaulty processor* P *increments its clock to be* 1 *without tossing a coin, then just before* $\mathcal{P}$ *all the nonfaulty processors' clock values were* 0.

Proof: The variable *last_increment* is assigned during every pulse. Thus, since $\mathcal{P}$ follows the first pulse, P indeed increments during $\mathcal{Q}$, the pulse before $\mathcal{P}$. Thus by Lemma 3.1 all the nonfaulty processors have clock values 0 after $\mathcal{Q}$ and before $\mathcal{P}$. ■

The next theorem uses the scheduler-luck game of [DIM-91, DIM-95] to analyze the randomized protocol. The scheduler-luck game has two competitors, *scheduler* (adversary) and *luck*. The goal of the scheduler is to prevent the protocol from reaching a safe configuration while the goal of *luck* is to help the protocol reach a safe configuration. For the synchronous protocol a configuration is *safe* if for all nonfaulty processors, the logical clocks are equal and *last_increment* is true. For our system the scheduler chooses the message delays and clock drifts during the execution (within the predefined limitations). Each time the processor, activated by the scheduler, tosses a coin, *luck* may intervene and determine the result of the coin toss. It is proved in [DIM-91, DIM-95] that if, starting with any possible configuration c, *luck* has a strategy to win the scheduler-luck game within i interventions and expected time t, then the system reaches a safe configuration within expected time $t \cdot 2^i$. The main observation used for this proof is the fact that if a coin toss result differs from the desired result (according to *luck* strategy) a configuration is reached from which a new game can begin.

Theorem 3.5 *In expected* $M_{lc} \cdot 2^{2(n-f)}$ *pulses, the system reaches a configuration in which the value of every nonfaulty processor's clock is 1.*

Proof: The proof is by the use of Lemma 1 of [DIM-91] (Theorem 5 of [DIM-95]). We present a strategy for *luck* to win the scheduler-luck game with $2(n-f)$ interventions and within $M_{lc}+2\pi$ time. The strategy of *luck* is (1) wait for the first pulse to elapse. Thereafter, (2) *luck* waits till a pulse $\mathcal{P}$ in which a nonfaulty processor with clock value 0 receives $n-f$ clock values that are 0. This occurs within the next M_{lc} pulses (if it does not occur by then, there is at least one nonfaulty processor that does not assign 0 to its clock during M_{lc} successive pulses, which is impossible). In case (2.1) during this pulse all the nonfaulty processors are either tossing a coin or assigning 1 without tossing. Then *luck* intervenes at most $n-f$ times and fixes the coin toss results of all the nonfaulty processors to be 1. Otherwise, (2.2) if there is a nonfaulty processor P that is neither about to toss a coin nor about to assign 1 without tossing, then *luck* intervenes and fixes all the coin toss results (less than $n-f$) to be 0. Note that before $\mathcal{P}$, P's clock is not equal to 0. Thus, by Claim 3.4 no processor assigns 1 without tossing a coin. By Lemma 3.1 and the fact that some nonfaulty processor tosses a coin during $\mathcal{P}$, it holds that following $\mathcal{P}$ the clock values of all the nonfaulty processors are 0. Therefore, in the next pulse case (2.1) is reached and *luck* could intervene and fix at most $n-f$ coin toss results to ensure that the desired global state is reached. ■

By Theorem 3.5 the system reaches a configuration in which the value of every nonfaulty processor's clock is 1, in expected time $M_{lc} \cdot 2^{2(n-f)}$. It is easy to see that in any successive pulse, all the nonfaulty processors have the same clock value. Thus the clock agreement requirement holds with $\gamma = 0$. Since the clocks of the nonfaulty processors are incremented by 1 in every pulse and the pulses are constant time apart, the clock validity requirement also holds. Note that the clock value could be multiplied by π (if π is known), the time difference between two successive pulses, in order to yield a clock value that reflects real time. Otherwise, the value of a of the clock validity requirement encodes $1/\pi$.

3.2 Accelerating the Protocol

If $M_{lc} = 2^{64}$, our protocol converges after expected $2^{64} \cdot 2^{2(n-f)}$ synchronization pulses. Certainly, because of this time complexity this protocol cannot be used in practice. However, if M_{lc}, n, and f are all small[2] then the expected number of pulses required is reasonably small. For instance, if $M_{lc} = 2$, $n = 4$, and $f = 1$, then the expected number of pulses is 128. We use the above observation to accelerate our protocol. We achieve synchronization of clock values in the range of $M_{lc} = 2^{64}$ values within expected number of pulses that is less than $381 \cdot 2^{2(n-f)}$. (For $M_{lc} = 2^{16}$, synchronization occurs within expected number of pulses that is less than $58 \cdot 2^{2(n-f)}$ pulses).

We define the *Chinese remainder counter* by the use of the Chinese remainder theorem, which appears in [Kn-81] p. 270:

Theorem 3.6 *Let m_1, m_2, ..., m_r be positive integers that are relatively prime in pairs, i.e., $gcd(m_j, m_k)=1$ when $j \neq k$. Let $m = m_1 m_2 \cdots m_r$, and let a, u_1, u_2,...,u_r be integers. Then there is exactly one integer u that satisfies the conditions $a \leq u < a + m$, and $u \equiv u_j (modulo\ m_j)$ for $1 \leq j \leq r$.*

We use the Theorem for the case $a = 0$ and $m \geq M_{lc}$. Let $2, 3, 5, ..., p_j$ be the series of prime numbers up to the j-th prime such that $2 \cdot 3 \cdot 5 \cdot ... \cdot p_{j-1} < M_{lc} \leq 2 \cdot 3 \cdot 5 \cdot ... \cdot p_j$. We run j parallel versions of our protocol. The i-th version runs the protocol with $M_{lc} = p_i$. Each message carries the value of j clocks, one clock value for each version. The computation of the new clock value of some version i uses the values received for this particular version and is independent from the computation of all the other versions. Thus, the i-th version converges within expected $p_i \cdot 2^{2(n-f)}$ pulses. Therefore, the expected time for all the versions to be synchronized is less than $(p_1 + p_2 + \cdots + p_j) \cdot 2^{2(n-f)}$. This is an upper bound on the expectation since it corresponds to a scenario in which version i starts to synchronize after every version $k < i$ is already synchronized.

Now we apply the Chinese remainder theorem to show that every combination of those values is mapped to one and only one number in the range 0 to $2 \cdot 3 \cdot 5 \cdots p_j$. A well-known technique could be used in order to convert such a representation to its mapping (e.g., by Garner methods, c.f. p. 274 [Kn-81]).

The Chinese remainder theorem could be used for other implementations of distributed counters based on the number presentation method suggested in [ST-67]. One possible use is as a memory and communication efficient distributed counter. Let DC be a distributed counter that is maintained by a set of processors $P_1, P_2, \ldots, P_j$ that are triggered by a common pulse. P_i increments the counter mod p_i in every trigger. P_i does not need to store the entire bits of the clock or to send messages to indicate the carry (when its counter wraps around). Thus, when the counter is incremented no communication between processors is needed. Only when the value of the counter is to be scanned is communication required.

4 Semi-synchronous Protocol

In this section we drop the assumption of common pulses. We present a self-stabilizing randomized protocol for semi-synchronous systems. Due to space constraints, the formal description of the protocol and the full correctness proof are excluded from this section.

[2] It is reasonable to think of n and f as being small when a single processor can efficiently compute a task and additional processors are added only to ensure reliability. Let the *reliability* be $f/(n+f)$, the ratio of the number of faulty processors to the total number of processors. To reach a reliability of 0.25, the number of processors needed (and thus, in general terms, the blowup in the hardware and cost) is four. To improve the reliability to $2/7 \approx 0.28$ the blowup would be 7. Asymptotically, we need an infinite blowup to reach reliability of 1/3. Thus, most devices would use a relatively small number of processors for which our protocol stabilizes in a relatively short time.

Our protocol uses the fault-tolerant averaging function first introduced in [DL+-86] for solving approximate agreement and later used for clock synchronization in [WL-88]. Given a multiset of values, a processor applies the function by discarding the f highest and f lowest values and then taking the midpoint of the remaining values. It has been shown that this function, when used in the context of the protocols of [DL+-86, WL-88], approximately halves the range of values held by the nonfaulty processors.

In our situation, with bounded clocks, the notions of "highest" and "lowest" must be appropriately modified. But the real difficulty in directly applying the previous result is that the analysis showing the range is cut in half depends on all nonfaulty processors working with approximately the same multisets at each "round". The multisets can differ arbitrarily in the values corresponding to the faulty processors, but the values corresponding to nonfaulty processors must be close to the same (allowing for error introduced by clock drift and uncertain message delays). This "round" structure can be achieved because the actions of the processors are roughly synchronized in time in the [WL-88] protocols, due to the assumption of initial synchronization or of distinguished initial states.

Since our protocol is self-stabilizing, it cannot rely on either of those assumptions. Thus using the fault-tolerant averaging function in the obvious manner, with the processors starting with arbitrary information and collecting clock values at arbitrary times, would not ensure that the function is applied at the processors in rounds. For instance, P could apply the function to a multiset M, then subsequently Q could apply the function to a multiset M' that reflects P's new value instead of P's old value.

To achieve some sort of approximate rounds for applying the fault-tolerant averaging function, we first use randomization to bring all the clock values of the nonfaulty processors close to each other. Once this is achieved, all the nonfaulty processors collect (approximately) the same multisets from all the nonfaulty processors. In this stage the midpoint averaging function can be shown (cf. [WL-88]) to approximately halve the nonfaulty clock values, thus overcoming the ongoing effects of clock drift and uncertainty of message delay.

We now describe the protocol. A processor P_i has two synchronization procedures. The first is called the *averaging procedure* and the second is the *jumping procedure*. The averaging procedure is executed when the value of $clock_i$ is in a range greater than 0 and smaller than δ and T_a time has elapsed since the previous time that $clock_i$ had a value in this range. The jumping procedure is executed when T_j time has elapsed since the previous execution of the jumping procedure and P_i is not currently in the range dedicated for executing the averaging function. P_i measures T_a and T_j using its physical clock. Roughly speaking, the jumping procedure causes the clocks of the nonfaulty processors to be within a small range. Then the averaging procedure keeps the clocks of the nonfaulty processors in a small range by approximately halving the range each time the clock values wrap around.

Both the synchronization procedures of processor P_i start with a request for clock values. During the execution of the averaging procedure, a processor measures $2(d + \epsilon + \delta)$ time in order to make sure that all the requests for clock values arrive at their destinations and the responses return before it proceeds to decide on a new clock value. Thus each execution of the averaging procedure takes some period of time. We define the *symmetric clock* of $clock_i$ to be $clock_i + M_{lc}/2 \pmod{M_{lc}}$. In both procedures, if P_i finds $n - f$ clock values within a small range δ from $clock_i$, then P_i eliminates f values from each side of the symmetric clock value[3]. Then, in the jumping procedure, P_i chooses one of the clock values at random from the reduced clock values list, while in the averaging procedure, P_i chooses the midpoint of the reduced clock values list. In both procedures, if less than $n-f$ processors are found within δ from $clock_i$, P_i chooses randomly one of the clock values.

[3]For instance, if the collected values are 2,3,10,11, the symmetric clock value is 7 and $f = 1$, then 3 and 10 are eliminated.

4.1 Correctness Proof Sketch of the Semi-synchronous Protocol

A period of time is a *jumping period* if no nonfaulty processor executes the averaging procedure during this period. We choose T_a to be $2(n-f)(5T_j+d+\epsilon)(1+\rho)^2$. The next lemma proves that the above choice yields the existence of a period of length $5T_j(1+\rho)$ that is a jumping period.

Lemma 4.1 *Every T_a time there is a jumping period that is at least $5T_j(1+\rho)$ long.*

Proof: A processor measures time by the use of its physical clock, whose drift rate from real time is at most ρ. Thus, if a processor measures a period of time T on its physical clock, then the real time elapsed during the measurement is at least $T/(1+\rho)$ and at most $T(1+\rho)$. By the way T_a is chosen, in every period of length $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)^2/(1+\rho) = 2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)$, every nonfaulty processor executes the averaging function at most once. A processor measures $2(d+\epsilon+\delta)$ time in order to make sure that the requests for clock values arrive at their destinations and the responses arrive before it decides on a new clock value. Thus, the time that the averaging function is executed by each processor in a period of $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)$ is no more than $2(d+\epsilon+\delta)(1+\rho)$. Hence, the total time of averaging of all the processors during a period of $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)$ is no more than $(n-f)2(d+\epsilon+\delta)(1+\rho)$. Therefore, the total non averaging time is at least $2(n-f)(5T_j+d+\epsilon+\delta)(1+\rho)-(n-f)2(d+\epsilon+\delta)(1+\rho) = 2(n-f)5T_j(1+\rho)$. By the pigeon hole principle at least one jumping period is of length $2(n-f)5T_j(1+\rho)/(n-f+1) > 5T_j(1+\rho)$. ■

A *safe configuration* is a system configuration in which the nonfaulty processors' clocks are within $\delta/8$ of each other. Moreover, in case a processor is in the middle of collecting clock values then all the clock values in transit sent by nonfaulty processors are within this range too.

We use the following assumptions in our correctness proof:

Assumption 1: $(1+\rho)^2 < 6/5$, thus $\rho < 0.095$.

Assumption 2: $(n-f)\epsilon + 2T_j(1+\rho)\rho < \delta/8$.

Lemma 4.2 *During any jumping period of length $5T_j(1+\rho)$, with probability at least $1/n^{6(n-f)}$, the system reaches a safe configuration.*

Sketch of proof: We prove the lemma by presenting a sequence of random choice results, that forces the system to reach a configuration in which the clocks of all the nonfaulty processors are less than $\delta/8$ apart. This sequence of random choice results has probability of at least $1/n^{6(n-f)}$ to occur. Let c be the configuration at the beginning of the jumping period.

Without loss of generality we assume that the number of faulty processors f is the maximal possible[4] that does not violate the inequality $n > 3f$. Let c be the first configuration in a choosing period. For every nonfaulty processor P, *luck* counts the number of other nonfaulty processors that have clocks within $T_r = \delta + 4(T_j(\rho+\rho^2)) + \epsilon$ of P's clock in the configuration c. Each nonfaulty processor that has at least $n-2f-1$ such surrounding clock values is called an *anchor*.

We claim that all the anchor processors are at most $2T_r$ apart. Assume towards contradiction that there are two nonfaulty anchor processors, P and Q, such that their clock values are more than $2T_r$ apart. Thus, P is surrounded by $n-2f-1$ nonfaulty processors and Q is surrounded by $n-2f-1$ *different* nonfaulty

[4]In case there are fewer faulty processors, one could assume that some of the nonfaulty processors "only behave" like nonfaulty processors.

processors. Therefore, the total number of nonfaulty processors is at least $2(n-2f) = 2n-4f > n-f$, contradiction.

Note that it is possible that no anchor processor exists. In this case *luck* chooses one nonfaulty processor to be an anchor processor.

Then *luck* chooses a single anchor processor A out of the anchor processors.

Until every nonfaulty processor executes the jump procedure twice *luck* uses the following strategy: Every time a processor, P_j, chooses a clock value and the value of the clock of A is a possible choice (i.e., either P_j does not find $n-f$ within δ range or A is in the reduced clock values list), this value is chosen; otherwise the value of $clock_j$ is not changed. Let c_1 be the first configuration reached from c after each processor executes the jump procedure at least twice with results according to the strategy of *luck*. Let E_1 be the execution that starts with c and ends with c_1. Since in a jumping period every nonfaulty processor chooses a clock value at least once in every period of length $T_j(1+\rho)$, c_1 occurs at most $2T_j(1+\rho)$ time after c.

We now show that in c_1 all the nonfaulty processors are within $2T_r + 2T_j(1+\rho)2\rho$ of each other. We first show that any nonanchor processor, P, assigns the value of A's clock to P's clock either in the first execution of the jump procedure or in the second one. Every processor collects the clock values during every execution of the jump procedure. In particular a nonanchor processor, P_j, receives the value of the clock of A before the second execution of the jump procedure. Next we show that, in the second execution of the jump procedure P_j can choose the value of A's clock.

The choice of P_j is restricted to a subset of the clock values that P_j read, only if P_j finds $n-f$ clock values within δ range of $clock_j$. Since P_j is a nonanchor processor it holds in c_1 that there are less than $n-f$ processors within δ range of $clock_j$. Moreover, no nonfaulty processor can assign a clock value within δ range of $clock_j$ since: (1) Every nonfaulty processor P_k that changes its clock value by the use of the jump procedure assigns the clock value of A (with up to ϵ range from the clock of A). (2) Every nonfaulty processor P_k that does not change its clock value by the use of the jump procedure can have a rate of drift from the clock of P_j of at most 2ρ. Thus, the difference between $clock_j$ and $clock_k$ can be shorten by at most $2T_j(1+\rho)2\rho = 4(T_j(\rho+\rho^2))$. Thus, if P_k was more than $T_r = \delta + 4(T_j(\rho+\rho^2)) + \epsilon$ apart from P_j in c then P_j cannot consider P_k to have a clock in δ range from $clock_j$ during E_1 (unless P_j assigns $clock_j$ by the value of the clock of A).

This proves that in c_1 all the nonanchor processors are within $\epsilon + 4T_j\rho$ from A's clock. The anchor processors that do not assign the clock value of A to their clock during E_1 were at most $2T_r$ apart in c, thus they are at most $2T_r + 2T_j(1+\rho)2\rho$ apart in c_1.

The fact that all the nonfaulty processors are within a small range of each other is used to define a new anchor processor A'. A' is the nonfaulty processor left after removing f nonfaulty processors with the highest clock values (mod M_{lc}) and f nonfaulty processors with the smallest values (mod M_{lc}).

¿From c_1 and until every processor executes the jump procedure at least twice, *luck* continues as follows: Any processor P_i that is in the process of collecting clock values in c_1 does not change $clock_i$ in the first execution of the jump procedure. For any other execution of the jump function, *luck* intervenes to fix the result to be the clock of A' or a clock of a processor that has already set its clock to the value of A''s clock since c_1. We have to prove that the above is a possible result of the jump function. This is obvious when the processor does not find $n-f$ processors within δ from its clock, since the choice is not restricted. It is also clear for the first set of processors that execute the jump procedure and use the clock values in c_1 as the base for the decision on the new clock value. Moreover, since *luck* intervenes and fixes all those results to be the value of the clock of A', the reduced list of every processor that uses the new clock values includes either the clock of A' or a clock of a processor that assigned its clock by the clock of A'.

Hence, in the first configuration, c_2, that follows the first two executions of the jump function of all the processors following c_1, all the nonfaulty processors are within $(n-f)\epsilon + 2T_j(1+\rho)\rho$ of each other, which, by assumption 2, is less than $\delta/8$.

Following c_2 any processor that is waiting for answers in the process of collecting clocks does not change its clock value. Thus, $(d+\epsilon)(1+\rho)$ time after c_2 a safe configuration is reached.

The length of the execution is $2T_j(1+\rho)$ until c_1 is reached, $2T_j(1+\rho)$ from c_1 to c_2 and additional $(d+\epsilon)(1+\rho)$ until a safe configuration is reached. Thus, a safe configuration is reached following $(4T_j + d + \epsilon)(1+\rho) < 5T_j(1+\rho)$ from c. By Assumption 1, $5T_j(1+\rho) < 6T_j/(1+\rho)$. Thus any processor could choose at most six times in such a range. Thus the total number of interventions is $6(n-f)$. ■

Lemma 4.3 *In any configuration of any execution that starts with a safe configuration, the clock values of all the nonfaulty processors are within at most $\delta/2$ of each other.*

The main observation made for the proof of the above lemma is that starting in a safe configuration every processor that either executes the jumping or the averaging procedure finds $n-f$ clock values within δ from its clock value. Thus, the new clock value chosen when jumping or averaging is in the range of clock values of the nonfaulty processors. The averaging procedure approximately halves the range of the clock values of the nonfaulty processors whenever they pass the zero clock value.

Theorem 4.4 *In expected $O(T_a n^{6(n-f)})$ time the system stabilizes.*

5 Concluding Remarks

Extensive research has been done to find efficient clock synchronization protocols in the presence of Byzantine faults. In this work we considered a more severe (and realistic) model of faults, i.e., one that takes into account transient faults as well as Byzantine faults. When arbitrary corruption of state is possible, as is often the case with transient faults, it is no longer reasonable to approximate unbounded clocks with bounded clocks, no matter how large. Consequently, clocks that can take on only a bounded number of values (and wrap around when appropriate) have been assumed in this paper. We presented two randomized self-stabilizing protocols for synchronizing bounded clocks in the presence of f Byzantine processor failures, where $n > 3f$.

We believe that our observations and definitions for the types of faults to be considered and the type of clocks (namely, bounded) reflect reality and open new directions for research. Protocols designed under our fault tolerance model are more robust than existing clock synchronization protocols. Therefore, such protocols might be preferred by the system implementer over protocols that cope with *only* Byzantine faults.

Acknowledgment: Many thanks to Brian Coan, Injong Rhee and Swami Natarajan for helpful discussions.

References

[Dij-74] E. W. Dijkstra, "Self stabilizing systems in spite of distributed control," *Communication of the ACM*, vol. 17, 1974, pp. 643–644.

[DHS-86] D. Dolev, J. Y. Halpern, and H. R. Strong, "On the possibility and impossibility of achieving clock synchronization," *Journal of Computer and Systems Science,* vol. 32, no. 2, 1986, pp. 230–250.

[DIM-91] S. Dolev, A. Israeli and S. Moran, "Uniform dynamic self stabilizing leader election," *Proc. of the 5th International Workshop on Distributed Algorithms*, 1991, pp. 167–180.

[DIM-95] S. Dolev, A. Israeli, and S. Moran, "Analyzing Expected Time by Scheduler-Luck Games," *IEEE Transactions on Software Engineering*, vol. 21, no. 5, May 1995.

[DL+-86] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *Journal of the ACM,* vol. 33, 1986, pp. 499–516.

[DW-93] S. Dolev and J. L. Welch, "Wait-free clock synchronization," *Proc. of the Twelfth ACM Symp. on Principles of Distributed Computing,* 1993, pp. 97–108.

[GP-93] A. S. Gopal and K. J. Perry, "Unifying self-stabilization And fault-tolerance," *Proc. of the Twelfth ACM Symp. on Principles of Distributed Computing,* 1993, pp. 195–206.

[HS+-84] J. Halpern, B. Simons, R. Strong, and D. Dolev, "Fault-tolerant clock synchronization," *Proc. of the Third ACM Symp. on Principles of Distributed Computing,* 1984, pp. 89–102.

[Kn-81] D. E. Knuth, *The art of computer programming*, Vol. 2, 2nd edition, Addison-Wesley, 1981.

[LM-85] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, no. 1, 1985, pp. 1–36.

[LSP-82] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem," *ACM Trans. on Prog. Lang. and Sys.*, vol. 4, no. 3, July 1982, 382-401.

[MS-85] S. Mahaney and F. Schneider, "Inexact agreement: accuracy, precision and graceful degradation," *Proc. of the Fourth ACM Symp. on Principles of Distributed Computing,* 1985, pp. 237–249.

[RSB-90] P. Ramanathan, K. G. Shin, and R. W. Butler, "Fault-tolerant clock synchronization in distributed systems," *IEEE Computer*, October, 1990, pp. 33–42.

[ST-87] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, 1987, pp. 626–645.

[ST-67] S. Szabo, and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*, McGraw-Hill, 1967.

[WL-88] J. L. Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation,* vol. 77, no. 1, 1988, pp. 1–36.