Clock Synchronization with Faults and Recoveries

(Extended Abstract)

Boaz Barak*
The Weizmann Institute of Science
boaz@wisdom.weizmann.ac.il

Shai Halevi
IBM Watson Research Center
shaih@watson.ibm.com

Amir Herzberg
IBM Haifa Research Lab
amir@il.ibm.com

Dalit Naor *
IBM Almaden Research Center
dalit@almaden.ibm.com

ABSTRACT

We present a convergence-function based clock synchronization algorithm, which is simple, efficient and fault-tolerant. The algorithm is tolerant of failures and allows recoveries, as long as less than a third of the processors are faulty 'at the same time'. Arbitrary (Byzantine) faults are tolerated, without requiring awareness of failure or recovery. In contrast, previous clock synchronization algorithms limited the total number of faults throughout the execution, which is not realistic, or assumed fault detection.

The use of our algorithm ensures secure and reliable time services, a requirement of many distributed systems and algorithms. In particular, secure time is a fundamental assumption of *proactive secure* mechanisms, which are also designed to allow recovery from (arbitrary) faults. Therefore, our work is crucial to realize these mechanisms securely.

Keywords

Clock synchronization, Mobile adversary, Proactive systems

*Work done while at IBM Haifa.

1. INTRODUCTION

Accurate and synchronized clocks are extremely useful to coordinate activities between cooperating processors, and therefore essential to many distributed algorithms and systems. Although computers usually contain some hardware-based clock, most of these are imprecise and have substantial drift, as highly precise clocks are expensive and cumbersome. Furthermore, even hardware-based clocks are prone to faults and/or malicious resetting. Hence, a clock synchronization algorithm is needed, that lets processors adjust their clocks to overcome the effects of drifts and failures. Such algorithm maintains in each processor a logical clock, based on the local physical clock and on messages exchanged with other processors. The algorithm must deal with communication delay uncertainties, clock imprecision and drift, as well as link and processor faults.

In many systems, the main need for clock synchronization is to deal with faults rather than with drift, as drift rates are quite small (some works e.g. [11, 12] actually ignore drifts). It should be noted, however, that clock synchronization is an on-going task that never terminates, so it is not realistic to limit the total number of faults during the system's lifetime. The contribution of this work is the ability to tolerate unbounded number of faults during the execution, as long as 'not too many processors are faulty at once'. This is done by allowing processors which are no longer faulty to synchronize their clocks with those of the operational processors.

Out protocol withstands arbitrary (or Byzantine) faults, where affected processors may deviate from their specified algorithm in an arbitrary manner, potentially cooperating maliciously to disrupt the goal of the algorithm or system. It is obviously critical to tolerate such faults if the system is to be secure against attackers, i.e. for the design of secure systems. Indeed, many secure systems assume the use of synchronized clocks, and while usually the effect of drifts can be ignored, this assumption may become a weak spot exploited by an attacker who maliciously changes clocks. Therefore, solutions frequently try not to rely on synchronized clocks, e.g. use instead freshness in authentication protocols (e.g. Kerberos [22]). However, this is not always achievable as often synchronized clocks are essential for efficiency or functionality. In fact, some security tasks require

securely synchronized clocks by their very definition, for example time-stamping [14] and e-commerce applications such as payments and bids with expiration dates. Therefore, secure time services are an integral part of secure systems such as DCE [25], and there is on-going work to standardize a secure version of the Internet's Network Time Protocol in the IETF [28]. (Note that existing 'secure time' protocols simply authenticate clock synchronization messages, and it is easy to see that they may not withstand a malicious attack, even if the authentication is secure.)

The original motivation for this work came from the need to implement secure clock synchronization for a *proactive security* toolkit [1]: Proactive security allows arbitrary faults in any processor - as long as no more than f processors are faulty during any (fixed length) period. Namely, proactive security makes use of processors which were faulty and later recovered. It is important to notice that in some settings it may be possible for a malicious attacker to avoid detection, so a solution is needed that works even when there is no indication that a processor failed or recovered. To achieve that, algorithms for proactive security periodically perform some 'corrective/maintenance' action. For example, they may replace secret keys which may have been exposed to the attacker. Clearly, the security and reliability of such periodical protocols depend on securely synchronized clocks, to ensure that the maintenance protocols are indeed performed periodically. There is substantial amount of research on proactive security, including basic services such as agreement [24], secret sharing [23, 17], signatures e.g. [16] and pseudo-randomness [4, 5]; see survey in [3]. However, all of the results so far assumed that clocks are synchronized. Our work therefore provides a missing foundation to these and future proactive security works.

1.1 Relations to prior work

There is a very large body of research on clock synchronization, much of it focusing on fault-tolerance. Below we focus on the most relevant works.

A number of works focus on handling processor faults, but ignore drifts. Dolev and Welch [11, 12] analyzed clock synchronization under a hybrid faults model, with recovery from arbitrary initial state of all processors (self stabilization) as well as napping (stop) failures in any number of processors in [11] or Byzantine faults in up to third of the processors in [12]. Both works assume a synchronous model, and synchronize logical clocks - the goal is that all clocks will have the same number at each pulse. Our results are not directly comparable, since it is not clear if our algorithm is self stabilizing. (In our analysis we assume that the system is initialized correctly.) On the other hand, we allow Byzantine faults in third of the processors during any period, work in asynchronous setting, allow drift and synchronize to real time.

The model of time-adaptive self stabilization as suggested by Kutten and Patt-Shamir [18] is closer to ours; there, the goal is to recover from arbitrary faults at f processors in time which is a function of f. We notice this is a weaker model than ours in the sense that it assumes periods of no faults. A time-adaptive, self-stabilizing clock synchronization protocol, under asynchronous model, was presented by Herman [15]. This protocol is not comparable to ours as it does not allow drifts and does not synchronize to real time.

Among the works dealing with both processor faults and drifts, most assume that once a processor failed, it never recovers, and that there is a bound f on the number of failed processors throughout the lifetime of the system. Many such works are based on local convergence functions. An early overview of this approach can be found in Schneider's report [26]. A very partial list of results along this line includes [13, 7, 8, 9, 21, 2, 20]. The Network Time Protocol, designed by Mills [21], allows recoveries, but without analysis and proof. Furthermore, while authenticated versions of [21] were proposed, so far these do not attempt recovery from malicious faults.

Our algorithm uses a convergence function similar to that of Fetzer and Cristian [9] (which, in turn, is a refinement of that of Welch and Lynch [20]). However, it seems that one of the design goals of the solution in [9] is incompatible with processor recoveries. Specifically, [9] try to minimize the change made to the clocks in each synchronization operation. Using such small correction may delay the recovery of a processor with a clock very far from the correct one (with [9] such recovery may never complete). This problem accounts for the difference between our convergence function and the one in [9]. In the choice between small maximum correction value, and fast recovery time, we chose the latter.

Another aspect in which [9] is optimal, is the maximum logical drift (see Definition 3 in Section 2.3). In their solution, the logical drift equals the hardware drift, whereas in our solution there is an additive factor of $O(2^{-K})$, where K is the number of synchronization operations performed in every time period. (Roughly, we assume that less than a third of the processors are faulty in each time period, and require that several synchronization operations take place in each such period.) As our model approaches that of [9] (i.e., as the length of the time period approaches infinity), this added factor to the logical drift approaches zero. We conjecture that 'optimal' logical drift can not be achieved in the mobile faults model.

Another difference between our algorithm and several traditional convergence function based clock synchronization algorithms, is that many such solutions proceed in *rounds*, where correct processes approximately agree on the time when a round starts. At the end of each round each processor reads all the clocks and adjusts its own clock accordingly. In contrast to this, our protocol (and also NTP [21]) do not proceed in rounds. We believe that implementing round synchronization across a large network such as the Internet could be difficult.

A previous work to address faults and recoveries for clock synchronization is due to Dolev, Halpern, Simons and Strong [10]. In that work it is assumed that faults are *detected*. In practice, faults are often undetected - especially malicious faults, where the attacker makes every effort to avoid detection of attack. Handling undetected faults and recoveries is critical for (proactive) security, and is not trivial, as a recovering processor may have its clock set to a value 'just a bit' outside the permitted range. The solution in [10] rely

on signatures rather than authenticated links, and therefore also limit the power of the attacker by assuming it cannot collect too many 'bad' signatures (assumption A4 in [10]).

The algorithm of Dolev *et al.* [10] is based on broadcast, and require that all processors sign and forward messages from all other processors. This has several practical disadvantages compared to local convergence function based algorithms such as in the present paper. Some of these disadvantages, which mostly result from its 'global' nature, are discussed by Fetzer and Cristian in [13]. Additional practical disadvantages of broadcast-based algorithms include sensitivity to transient delays, inability to take advantage of realistic knowledge regarding delays, and the overhead and delay resulting from depending on broadcasts reaching the entire network (e.g. the Internet). On the other hand, being a broadcast based algorithm , Dolev *et al.* [10] require only a majority of the processors to be correct (we need two thirds). Also [10] only requires that the subnet of non faulty processors be connected, rather than demanding a direct link between any two processors. (But implementing the broadcast used by [10] has substantial overhead and requires two-third of processors to be correct and connected.)

1.2 Informal Statement of the Requirements

A clock synchronization algorithm which handles faults and recoveries should satisfy:

SYNCHRONIZATION Guarantee that at all times, the clock values of the non-faulty processors are close to each other.[1]

ACCURACY Guarantee that the clock rates of non-faulty processors are close to that of the real-time clock. One reason for this requirement is that in practice, the set of processors is not an island and will sometimes need to communicate and coordinate with processors from the "outside world".

RECOVERY Guarantee that once a processor is no longer faulty, this processor recovers the correct clock value and rejoins the "good processors" within a fixed amount of time.

We present a formalization of this model and goals, and a simple algorithm which satisfies these requirements. We analyze our algorithm in a model where an attacker can temporarily corrupt processors, but not communication links. It may be possible to refine our analysis to show that the same algorithm can be used even if an attacker can corrupt both processors and links, as long as not too many of either are corrupted "at the same time".

2. FORMAL MODEL

2.1 Network and Clocks

Our network model is a fully connected communication graph of n processors, where each processor has its own local clock. We denote the processor names by $1, 2, \ldots n$ and assume that each processor knows its name and the names of all its neighbors. In addition to the processors, the network model also contains an adversary, who may occasionally corrupt processors in the network for a limited time. Throughout the discussion below we assume some bound ρ on the clock drift between "good processors", and a bound λ on the time that it takes to send a message between two good processors. We refer to ρ as the *drift bound* and to λ as the *message delivery bound*.

We envision the network in an environment with *real time*. A convenient way of thinking about the real time is as just another clock, which also ticks more or less at the same rate as the processors' clocks. For the purpose of analysis, it is convenient to view the local clock of a processor p as consisting of two components. One is an unresettable *hardware clock* H_p, and the other is an *adjustment variable* adj_p, which can be reset by the processor. The clock value of p at real time τ, denoted $C_p(\tau)$ is the sum of its hardware clock and adjustment factor at this time, $C_p(\tau) = H_P(\tau) + \mathsf{adj}_p$ (these are the same notations as in [26]).[2] We stress that H_p and adj_p are merely a mathematical convenience, and that the processors (and adversary) do not really have access to their values. Formally, the only operations that processor p can perform on H_p and adj_p are reading the value $H_p(\tau) + \mathsf{adj}_p$ and adding an arbitrary factor to adj_p. Other than these changes, the value of H_p changes continuously with τ (and the value of adj_p remains fixed).

DEFINITION 1 (CLOCKS). *The* hardware clock *of a processor p is a smooth, monotonically increasing function, denoted $H_p(\tau)$. The* adjustment factor *of p is a discrete function $\mathsf{adj}_p(\tau)$ (which only changes when p adds a value to the its adjustment variable). The* local clock *of p is defined as*

$$C_p(\tau) \stackrel{def}{=} H_p(\tau) + \mathsf{adj}_p(\tau) \tag{1}$$

We assume an upper bound ρ on the drift rate between processors' hardware clocks and the real time. Namely, for any $\tau_1 < \tau_2$, and for every processor p in the network, it holds that

$$(\tau_2 - \tau_1)/(1+\rho) \;\leq\; H_p(\tau_2) - H_p(\tau_1) \;\leq\; (\tau_2 - \tau_1)\cdot(1+\rho) \tag{2}$$

We note that in practice, ρ is usually fairly small (on the order of 10^{-6}).

2.2 Adversary Model

As we said above, our network model comes with an *adversary*, who can occasionally *break into* a processor, resetting its local clock to an arbitrary value. After a while, the adversary may choose to *leave* that processor, and then we would like this processor to *recover* its clock value.

We envision an adversary who can see (but not modify) all the communication in the network, and can also break into processors and leave them at wish. When breaking into a

[1] Note that a trivial solution of setting all local clocks to a constant value achieves the synchronization goal. The accuracy requirement prevents this from happening.

[2] In general adj_p does not have to be a discrete variable, and it could also depend on τ. We don't use that generality in this paper, though.

processor p, the adversary learns the current internal state of that processor. Furthermore, from this point and until it leaves p, the adversary may send messages for p, and may also modify the internal state of p, including its adjustment variable adj_p. Once the adversary leaves a processor p, it has no more access to p's internal state. We say that p is *faulty* (or *controlled* by the adversary), if the adversary broke into p and did not leave it yet.

We assume reliable and authenticated communication between processors p and q that are not faulty. More precisely, let λ denote the message delivery bound. Then for any processors p and q not faulty during $[\tau, \tau+\lambda]$, if p sends a message to q at time τ, then q receives exactly the same message from p during $[\tau, \tau+\lambda]$. Furthermore, if a non-faulty processor q receives a message from processor p at time τ, then either p has sent exactly this message to q during $[\tau-\lambda, \tau]$, or else it was faulty at some time during this interval.[3]

The power of the adversary in this model is measured by the number of processors that it can control within a time interval of a certain length. This limitation is reasonable because otherwise, even an adversary that can control only one processor at a time, can corrupt all the clocks in the system by moving fast enough from processor to processor.

DEFINITION 2 (LIMITED ADVERSARY). *Let $\theta > 0$ and $f \in \{1, 2, \ldots, n\}$ be fixed. An adversary is f-limited (with respect to θ) if during any time interval $[\tau, \tau+\theta]$, it controls at most f processors.*

We refer to θ as the time period *and to f as the* number of faulty processors*.*

Notice that Definition 2 implies in particular that an f-limited adversary who controls f processors and wants to break into another one, must leave one of its current processors at least θ time units before it can break into the new one. In the rest of the paper we assume that $n \geq 3f+1$.

2.3 Clock Synchronization Protocols

Intuitively, the purpose of a clock synchronization algorithm is to ensure that processors' local clocks remain close to each other and close to the real time, and that faulty processors become synchronized again quickly after the adversary leave them. It is clear, however, that no protocol can achieve instantaneous recovery, and we must allow processors some time to recover. Typically we want this recovery time to be no more than θ, so by the time the adversary breaks into the new processors, the ones that it left are already recovered.

DEFINITION 3 (CLOCK SYNCHRONIZATION). *Consider a clock synchronization protocol π that is executed in a network with drift rate ρ and message delivery bound λ, and in the presence of an f-limited adversary with respect to time period θ.*

[3] This formulation of "good links" does not completely rule out replay of old messages. This does not pause a problem for our application, however.

i. We say that π ensures synchronization *with maximal deviation δ, if at any time τ and for any two processors p, q not faulty during $[\tau-\theta, \tau]$, it holds that $|C_p(\tau) - C_q(\tau)| \leq \delta$.*

ii. We say that π ensures accuracy *with maximal drift $\tilde{\rho}$ and maximal discontinuity μ, if whenever p is not faulty during an interval $[\tau_1-\theta, \tau_2]$ (with $\tau_1 \leq \tau_2$), it holds that*

$$\begin{array}{l} C_p(\tau_2) - C_p(\tau_1) \;\geq\; (\tau_2-\tau_1)/(1+\tilde{\rho}) \;-\mu \\ \textit{and} \\ C_p(\tau_2) - C_p(\tau_1) \;\leq\; (\tau_2-\tau_1)\cdot(1+\tilde{\rho}) \;+\mu \end{array} \qquad (3)$$

3. A CLOCK SYNCHRONIZATION PROTOCOL

As in most (practical) clock synchronization protocols, the most basic operation in our protocol is the estimation by a processor of its peers' clocks. We therefore begin in Subsection 3.1 by discussing the requirements from a clock estimation procedure and describing a simple (known) procedure for doing that. Then, in Subsection 3.2 we describe the clock synchronization protocol itself. In this description we abstract the clock estimation procedure, and view it a "black box" that provides only the properties that were discussed before. Finally, in Subsection 3.3 we elaborate on some aspects of our protocol, and compare it with similar synchronization protocols for other models.

3.1 Clock Estimation

Our protocol's basic building block is a subroutine in which a processor p estimates the clock value of another processor q. The (natural) requirements from such a procedures are

Accuracy. The value returned from this procedure should not be too far from the actual clock value of processor q.

Bounded error. Along with the estimated clock value, p also gets some upper bound on the error of that estimation.

For technical reasons it is also more convenient to have this procedure return the *distance* between the local clocks of p and q, rather than the clock value of q itself. Hence we define a clock estimation procedure as a two-party protocol, such that when a processor p invokes this protocol, trying to estimate the clock value of another processor q, the protocol returns two values (d_q, a_q) (for distance and accuracy). These values should be interpreted as "since the procedure was invoked, there was a point in which the difference $C_q - C_p$ was about d_q, up to an error of a_q". Formally, we have

DEFINITION 4. *We say that a clock estimation routine has reading error λ and timeout* MaxWait*, if whenever a processor p is non-faulty during time interval $[\tau, \tau+\mathsf{MaxWait}]$, and it calls this routine at time τ to estimate the clock of q, then the routine returns at time $\tau' \leq \tau+\mathsf{MaxWait}$, with some values (d, a). Moreover, if q was also non-faulty during the interval $[\tau, \tau']$, then the values (d, a) satisfy the following:*

- *$a \leq \lambda$, and*

- *There was a time* $\tau'' \in [\tau, \tau']$ *at which* $C_q(\tau'') - C_p(\tau'') \in [d - a, d + a]$.

We now describe a simple clock estimation algorithm. The requestor p sends a message to q, who returns a reply to p containing the time according on the clock of q (when sending the reply). If p does not receive a reply within $\mathsf{MaxWait} = 2\lambda$ time (where λ is the message delivery bound), p aborts the estimation and sets $d_q = 0$ and $a_q = \infty$. Otherwise, if p sends its "ping" message to q at local time S, and receives an answer C at local time R, it sets $d_q = C - \frac{R+S}{2}$ and $a_q = \frac{R-S}{2}$.

Intuitively, p estimates that at its local time $\frac{R+S}{2}$, q's time was C. If the network is totally symmetric, that is the time for the message to arrive was identical on the way from p to q and on the way back from q to p, and p's clock progressed between S and R at constant rate, then the estimation would be totally accurate. In any case, if q returned an answer C, then at some time between p local time S and p local time R, q had the value C, so the estimation of the offset can't miss by more than $\frac{R-S}{2}$.

This simple procedure can be "optimized" in several ways. A common method, which is used in practice to decrease the error in estimating the peer's clock (at the expense of worse timeliness), is to repeatedly ping the other processor and choose the estimation given from the ping with the least round trip time. This is used, for example, in the NTP protocol [21].

Also to reduce network load it may be possible to piggyback clock querying messages on other messages, or to perform them in a different thread which will spread them across a time interval. Of course, if we implement the latter idea in the mobile adversary setting, a clock synchronization protocol should periodically check that this thread exists and restart it otherwise (to protect against the adversary killing that thread). We note that when implemented this way, we cannot guarantee the conditions of Definition 4 anymore, since the separate thread may return an old cached value which was measured before the call to the clock estimation procedure. (Hence, the analysis in this paper cannot be applied "right out of the box" to the case where the time estimation is done in a separate thread.)

3.2 The Protocol

SYNC is our clock synchronization protocol. It uses a clock estimation procedure such as the one described in Section 3.1, which we denote by *estimateOffset*, with the time-out bound denoted by MaxWait and maximal error λ. Other parameters in this protocol are the (local) time SyncInt between two executions of the synchronization protocol, and a parameter WayOff, which is used by a processor to gauge the distance between its clock and the clocks of the other processors. These parameters are (approximately) computed from the network model parameters ρ, λ and θ. The constraints that these parameters should satisfy are:

$\mathsf{SyncInt} \geq 2\mathsf{MaxWait} \geq 4\lambda$

$\mathsf{WayOff} \geq \delta + \lambda$, where δ is the maximum deviation we want to achieve (and we have $\delta > 16\lambda$).

These settings are further discussed in the analysis (Section 4.2) and in Section 3.3.

The SYNC protocol is described in Figure 1. The basic idea is that each processor p uses *estimateOffset* to get an estimate for the clocks of its peers. Then p eliminates the f smallest and f largest values, and use the remaining values to adjust its own clock. Roughly, p computes a "low value" C^m which is the $f+1$'st smallest estimate, and a "high value" C^M which is the $f+1$'st largest estimate. If p's own clock C_p is more than WayOff away from the interval $[C^m, C^M]$, then p knows that its clock is too far from the clocks of the "good processors", so it ignores its own clock and resets it to $(C^m + C^M)/2$. Otherwise, p's clock is "not too far" from the other processors, so we would like to limit the change to it. In this case, instead of completely ignoring the old clock value, p resets its clock to $(\min(C^m, C_p) + \max(C^M, C_p))/2$. (So if p's clock was below C^m or above C^M, it will only move half-way towards these values.)

The details of the SYNC protocol are slightly different, though, specifically in the way that the "low value" and "high value" are chosen. Processor p first uses the error bounds to generate overestimates and underestimates for these clock values, and then computes the "low value" C^m as the $f+1$'st smallest overestimate, and the "high value" the $f+1$'s largest underestimate. In the analysis we also assume that all the clock estimations are done in parallel, and that the time that it takes to make the local computations is negligible, so a run of SYNC takes at most MaxWait time on the local clock. (This is not really crucial, but it saves the introduction of an extra parameter in the analysis.)

3.3 Discussion

Our SYNC protocol follows the general framework of "convergence function synchronization algorithms" (see [26]), where the next clock value of a processor p is computed from its estimates for the clock values of other processors, using a fixed, simple convergence function.[4]

No rounds. As mentioned in section 1.1, one notable differences between our protocol and other protocols that have been proposed in the literature is that many convergence function protocols (for example [8, 9]) proceed in *rounds*, where each processor keeps a different logical clock for each round. (A round is the time between two consecutive synchronization protocols.) In these protocols, if a processor is asked for a "round-i" clock when this processor is already in its $i+1$'st round, it would return the value of its clock "as if it didn't do the last synchronization protocol".

In contrast, in our SYNC protocol a processor p always responds with its current clock value. This makes the analysis of the protocol a little more complicated, but it greatly simplifies the implementation, especially in the mobile adversary setting (since variables such as the current round

[4] In the current algorithm and analysis, a processor needs to estimate the clocks of all other processors; we expect that this can be improved, so that a processor will only need to estimate the clocks of its local neighbors.

Figure 1: Algorithm SYNC for processor p

```
Parameters: SyncInt        // time between synchronizations
            WayOff         // bound for clocks which are very far from the rest

1. Every SyncInt time units call sync()

3. function sync () {
4.      For each q ∈ {1, ..., n} do
5.          (d_q, a_q) ← estimateOffset(q)
6.          d̄_q = d_q + a_q              // overestimate d_q
7.          d̲_q = d_q − a_q              // underestimate d_q
8.      m ← the f + 1'st smallest d̄_q
9.      M ← the f + 1'st largest d̲_q
10.     If m ≥ −WayOff and M ≤ WayOff
11.         then adj_p ← adj_p + (min(m, 0) + max(M, 0))/2
12.         else adj_p ← adj_p + (m + M)/2
13. }
```

number, last round's clock, and the time to begin the next round have to be recovered from a break in).

Known values. Another practical advantage of our protocol is that it does not require to know the values of parameters such as the message delivery bound λ , the hardware drift ρ, the maximum deviation δ, which may be hard to measure in practice (in fact they may even change during the course of the execution). We only use these values in the analysis of the protocol. In practice, all the algorithm parameters which do depend on these value (like MaxWait, SyncInt and WayOff) may overestimate them by a multiplicative factor without much harm (*i.e.* without introducing such a factor to the maximum deviation, logical drift or recovery time actually achieved).

When to perform SYNC? In our protocol, a processor executes the SYNC protocol every SyncInt time units of local time, and we do not make any assumptions about the relative times of SYNC executions in different processors. A common way to implement this is to set up an alarm at the end of each execution, and to start the new execution when this alarm goes off. In the mobile adversary setting, one must make sure that this alarm is recovered after a break-in.

We note that our analysis does not depend on the processors executing a SYNC exactly every SyncInt time units. Rather, all we need is that during a time interval of $(1+\rho)$(SyncInt+ MaxWait) real time, each processor completes at least one and at most two SYNC's.

4. ANALYSIS

Let T denote some value such that every non-faulty processor completes at least one and at most two full SYNC's during any interval of length T. Specifically, setting $T = (1+\rho)$SyncInt + 2MaxWait is appropriate for this purpose (where SyncInt is the time that is specified in the protocol, MaxWait is a bound on the execution time of a single SYNC, and ρ is the drift rate).

4.1 Main Theorem

The following main theorem characterizes the performance achieved by our protocol:

THEOREM 5. *Let T be as defined above, let $K \stackrel{def}{=} \lfloor \frac{\theta}{T} \rfloor$ and assume that $K \geq 5$. Then*

i. The SYNC *protocol fulfills the synchronization requirement with maximum deviation $\delta = 16\lambda + 18\rho T + 4C$ where $C = \frac{17\lambda + 18\rho T}{2^{K-3}}$.*

ii. The SYNC *protocol fulfills the accuracy requirement with logical drift $\tilde{\rho} = \rho + \frac{C}{2T}$ and discontinuity $\mu = \delta + C/2$.*

We note that the theorem shows a tradeoff between the rate at which the SYNC protocol is performed (as a function of θ) and how optimal its performance is. That is, if we choose T to be small compared to θ (for instance $T = \frac{\theta}{20}$) then C is very small and so we get almost perfect accuracy ($\tilde{\rho} \approx \rho$) and the significant term in the maximum deviation bound is 16λ.

4.2 Clock Bias

For the purpose of analysis, it will be more convenient to consider the *bias* of the clocks, rather than the clock values themselves. The bias of processor p at time τ is the difference between its logical clock and the real time, and is denoted by $B_p(\tau)$. Namely,

$$B_p(\tau) \stackrel{def}{=} C_p(\tau) - \tau \tag{4}$$

When the real-time τ is implied in the context, we often omit it from the notation and write just B_p instead of $B_p(\tau)$.

In the analysis below we view the protocol SYNC as affecting the biases of processors, rather than their clock values. In particular, in an execution of SYNC by processor p, we can view d_q as an estimate for $B_p - B_q$ rather than an estimate for $C_p - C_q$, and we can view the modification of adj_p in the last step as a modification of B_p. We can therefore re-write

Figure 2: Algorithm SYNC for processor p: bias formulation

```
Parameters: SyncInt        // time between sunchronizations
            WayOff         // Bound for clocks which are very far from the rest

1. Every SyncInt time units call sync()

3. function sync () {
4.      For each q ∈ {1, ..., n} do
5.          (d_i, a_i) ← estimateOffset(i)
6.          B̄_q = B_p + d_q + a_q          // overestimate B_q
7.          B̲_q = B_p + d_q − a_q          // underestimate B_q
8.      B_p^(m) ← the f + 1'st smallest B̄_q
9.      B_p^(M) ← the f + 1'st largest B̲_q
10.     If B_p − B_p^(m) ≤ WayOff and B_p^(M) − B_p ≤ WayOff
11.         then B_p ← ( min(B_p^(m), B_p) + max(B_p^(M), B_p) ) /2
12.         else B_p ← ( B_p^(m) + B_p^(M) ) /2
13. }
```

the protocol in terms of biases rather than clock values, as in Figure 2.

We note that by referencing B_p in the protocol, we mean $B_p(\tau)$ where τ is the real time where this reference takes place. We stress that the protocol cannot be implemented as it is described in Figure 2, since a processor p does not know its bias B_p. Rather, the above description is just an alternative view of the "real protocol" that is described in Figure 1.

4.3 Proof Overview of the Main Theorem

Below we provide only an informal overview of the proof. A few more details (including a useful piece of syntax and statements of the technical lemmas) can be found in Appendix A. A complete proof will be included in the full version of the paper. For simplicity, in this overview we only look at the case with no drifts and no clock-reading errors, namely $\rho = \lambda = 0$. (Note that in this case, we always have $a_q = 0$, so in Steps 6-7 of the protocol we get $\underline{B}_q = \overline{B}_q = B_p + d_q$.)

The analysis looks at consecutive time intervals $I_0, I_1, \ldots$, each of length T, and proceeds by induction over these intervals. For each interval I_i we prove "in spirit" the following claims:

i. The bias values of the "good processors" get closer together: If they were at distance δ from each other at the beginning of I_i, they will be at distance $7\delta/8$ at the end of it.

ii. The bias values of the "recovering processors" gets (much) closer to those of the good processors. If a recovering processor was at distance ξ from the "range of good processors" at the beginning of I_i, it would be at distance at most $\xi/2$ from that range at the end of I_i.

It therefore follows that after a few such intervals, the bias of a "recovering processor" will be at most δ away from those of the "good processors".

To prove the above claims, our main technical lemma considers a given interval I_i, and assumes that there is a set G of at least $n-f$ processors, which are all non-faulty throughout I_i, and all have bias values in some small range at the beginning of I_i (w.l.o.g., this can be the range $[-D, D]$). Then, we prove the following three properties:

Property 1 We first show that the biases of the processors in G remain in the range $[-D, D]$ throughout the interval I_i. This is so because in every execution of SYNC, a processor $p \in G$ always gets biases in that range from all other processors in G. Since G contains more than $2f$ processors, then both $B_p^{(m)}$ and $B_p^{(M)}$ are in that range, and so p's bias remains in that range also after it completes the SYNC protocol.

Also, if follows from the same argument that we always have $B_p - B_p^{(m)} \leq \mathsf{WayOff}$ and $B_p^{(M)} - B_p \leq \mathsf{WayOff}$, so processor p never ignores its own current bias in Step 11 of the protocol.

Property 2 Next we consider processors $p \in G$ whose initial bias values are low (say, below the median for G). Since p executes at most two SYNC's during the time interval I_i, and in each SYNC it takes the average of its own current bias and another bias below D, the bias of p remains bounded strictly below D. Specifically, one can show that the resulting bias values cannot be larger than $(Z + 3D)/4$ (where Z is the initial median value).

Similarly, for the processors $q \in G$ with high initial bias values, the bias values remain bounded strictly above $-D$, specifically at least $(Z - 3D)4$.

Property 3 Last, we use the result of the previous steps to show that at the end of the interval, the bias of every processor in G is between $(Z - 7D)/8$ and $(Z + 7D)/8$.

(Hence, in the case of no errors or drifts, the size of the interval that includes all the processors in G shrunk from $2D$ to $7D/4$.)

To see this, recall that by the result of the previous step, whenever a processor $p \in G$ executes a SYNC, it gets bias values which are bounded by $(Z+3D)/4$ from all the processors with low initial biases – and so its low estimate $B_p^{(m)}$ must also be smaller than $(Z+3D)/4$. Similarly, it gets bias values which are bounded above $(Z-3D)/4$ from all the processors with high initial biases – and so its high estimate $B_p^{(M)}$ must be larger than $(Z-3D)/4$. Since the bias of p after its SYNC protocol is computed as $(\min(B_p^{(m)}+B_p)+\max(B_p^{(M)}+B_p))/2$, and since B_p is in the range $[-D,D]$ by the result of the first step, the result of this step follows.

Moreover, a similar argument can be applied even to a processor outside G, whose initial bias is not in the range $[-D,D]$. Specifically, we can show that if at the beginning of interval I_i, a non-faulty processor p has high bias, say $D+\xi$ for some $\xi > 0$, then at the end of the interval the bias of p is at most $(Z+7D)/8 + \xi/2$. Hence, the distance between p and the "good range" shrinks from ξ to $\xi/2$.

A formal analysis, including the effects of drifts and reading errors, will be included in the full version of the paper.

5. FUTURE DIRECTIONS

Our results require that at most a third of the processors are faulty during each period. Previous clock synchronization protocols assuming authenticated channels (as we do) were able to require only a majority of non-faulty processors [19, 27]. It is interesting to close this gap. In [10] there is another weaker requirement: only that subnetwork containing non-faulty processors remain connected (but [10] also assumes signatures). It may be possible to prove a variant of this for our protocol, in particular it would be interesting to show that it is sufficient that the non-faulty processors form a sufficiently connected subgraph. If this holds, it will also justify limiting the clock synchronization links to a limited number of neighbors for each processor, which is one of the practical advantages of convergence based clock synchronization.

(It should be noted that $(3f+1)$-connectivity is not sufficient for our protocol. One can construct a graph on $6f+2$ nodes which is $(3f+1)$-connected, and yet our protocol does not work for it. This graph consists of two cliques of $3f+1$ nodes, and in addition the i'th node of one clique is connected to the i'th node of the other. Now, this graph is clearly $3f+1$ connected, but our protocol cannot guarantee that the the clocks in one cliques do not drift apart from those in the other.)

Additional work will be required to explore the practical potential of our protocol. In particular, practical protocols such as the Network Time Protocol [21] involve many mechanisms which may provide better results in typical cases, such as feedback to estimate and compensate for clock drift. Such improvements may be needed to our protocol (while making sure to retain security!), as well as other refinements in the protocol or analysis to provide better bounds and results in typical scenarios.

The Synchronization and Accuracy requirements we defined only talk about the behavior of the protocol when the adversary is suitably limited. It may also be interesting to ask what happens with stronger adversaries. Specifically, what happens if the adversary was "too powerful" for a while, and now it is back to being f-limited. An alternative way of asking the same question is what happens when the adversary is limited, but the initial clock values of the processors are arbitrary. Along the lines of [11, 12], it is desirable to improve the protocol and/or analysis to also guarantee *self stabilization*, which means that the network eventually converges to a state where the non-faulty processors are synchronized.

6. REFERENCES

[1] B. Barak, A. Herzberg, D. Naor and E. Shai, The Proactive Security Toolkit and Applications, *Proceedings of the 6th ACM Conference on Computer and Communication Security*, Nov. 1999, Singapore, Pages 18–27.

[2] A. Bouzelat and Z. Mammeri, Simple reading, implicit rejection and average function for fault-tolerant physical clock synchronization, *Proceedings of 23rd EuroMicro Conference*, Sep. 1997, pages 524–531.

[3] R. Canetti, R. Gennero, A. Herzberg and D. Naor, Proactive Security: Long-term protection against break-in, *CryptoBytes*, Vol. 3 No. 1, Spring 1997.

[4] R. Canetti and A. Herzberg, Maintaining Security in the Presence of Transient Faults, *Proceedings of Crypto' 94*, pages 425–438, August 1994.

[5] C.S. Chow and A. Herzberg, Network Randomization Protocol: A proactive pseudo-random generator, *Proceedings of 5th Usenix Unix Security Symposium*, Salt Lake City, Utah, June 1995, pages 55–63.

[6] R. Canetti, S. Halevi, and A. Herzberg. Maintaining authenticated communication in the presence of break-ins. *Journal of Cryptology*, to appear. Preliminary version appeared in *Proceedings of 16'th Annual ACM Symposium on Principles of Distributed Computing*, pages 15–24. ACM Press, 1997.

[7] F. Cristian, Probabilistic Clock Synchronization, *Distributed Computing*, Vol. 3, pp. 146–158, 1989.

[8] F. Cristian and C. Fetzer, Probabilistic Internal Clock Synchronization, *Proceedings of the 13th Symposium on Reliable Distributed Systems*, Oct. 1994, Dana Point, CA. Pages 22–31.

[9] F. Cristian and C. Fetzer, An Optimal Internal Clock Synchronization Algorithm, *Proceedings of the 10th Annual IEEE Conference on Computer Assurance*, June, 1995, Gaithersburg, MD. Pages 187–196.

[10] D. Dolev, J.Y. Halpern, B. Simons and R. Strong, Dynamic Fault-Tolerant Clock Synchronization, *JACM* Vol. 42, No. 1, Jan. 1995, pp. 143–185.

[11] S. Dolev and J. Welch, Wait-free clock synchronization, *Proceedings of 12'th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 97–108. ACM Press, 1993.

[12] S. Dolev and J. Welch, Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults, *Proc. of the 2nd Workshop on Self-Stabilizing Systems*, May 1995.

[13] C. Fetzer and F. Cristian, Lower bounds for convergence function based clock synchronization, *14th PODC*, Aug 1995, Ottowa, Canada, pp. 137–142.

[14] S. Haber and W.S. Stornetta How to Time-Stamp a Digital Document, *Journal of Cryptology*, 1991, Vol. 3, pages 99–111.

[15] T. Herman, Phase Clocks for Transient Fault Repair, Technical Report 99-08, University of Iowa Department of Computer Science, 1999.

[16] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk and M. Yung, Proactive public key and signature systems, *Proceedings of the 4th ACM Conference on Computer and Communication Security*, 1997.

[17] A. Herzberg, S. Jarecki, H. Krawczyk and M. Yung, Proactive Secret Sharing, or: How to cope with perpetual leakage, *Proceedings of Crypto' 95*, August 1995, pp. 339–352.

[18] S. Kutten and B. Patt-Shamir, Time-adaptive self stabilization, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 149–158, 1997.

[19] L. Lamport and P.M. Melliar-Smith, Synchronizing clocks in the presence of faults, *JACM* Vol. 32, No. 1, Jan. 1985, pp. 52–78.

[20] J. Lundelius-Welch and N. Lynch, A new fault-tolerant algorithm for clock synchronization, *Information and Computation*, Vol. 77, No. 1, pages 1–36, 1988.

[21] D.L. Mills, Internet time synchronization: the Network Time Protocol. In *IEEE Trans. Communications*, October 1991, pp. 1482- 1493.

[22] B.C. Neuman and T. Ts'o, Kerberos: An Authentication Service for Computer Networks, *IEEE Communications Magazine*, Sep. 1994, Vol. 32, No. 9, pp. 33–38.

[23] R. Ostovsky and M. Yung, How to withstand mobile virus attacks, *Proceedings of PODC*, 1991, pp. 51–61.

[24] R. Reischuk, A new solution to the byzantine generals problem, *Information and Control*, pages 23–42, 1985.

[25] W. Rosenberry, D. Kenney and G. Fisher, Understanding DCE, Chapter 7: DCE Time Service: Synchronizing Network Time, *O'Reilly*, Oct. 1992.

[26] Fred B. Schneider, Understanding Protocols for Byzantine Clock Synchronization *Technical Report TR87-859, CS Department, Cornell University*, 1987.

[27] T.K. Srikanth and S. Toueg, Optimal clock synchronization, *JACM* Vol. 34, No. 3, July 1987, pp. 626–645.

[28] IETF Secure Time (Stime) Working Group, http://www.stime.org/.

APPENDIX

A. SOME DETAILS

Below we describe the syntax that we use to formally prove Theorem 5, and states explicitly the technical lemmas that we prove.

A.1 Bias values and envelopes

It is useful to think of the bias values of processors as if they are drawn on a two-dimensional plane, where one axis is real time values and the other axis is bias values. (We call this the (τ, β)-plane.)

Note that since the drift of processors is bounded by ρ, the bias of a processor that does not reset its clock during a time interval of length τ cannot change by more than $\rho\tau$. Hence, if we know that the bias of a processor at time τ is in the interval $[a, b]$, and that this processor did not reset its clock between time τ and τ', then its bias at time τ' must be in the interval $[a - \rho(\tau' - \tau),\ b + \rho(\tau' - \tau)]$. This motivates the following definition of an *envelope* in the (τ, β)-plane.

DEFINITION 6. *An* envelope *in the (τ, β)-plane is a region of the form*

$$E = \{(\tau, \beta) \mid \tau \geq \tau_0,\ a - \rho(\tau - \tau_0) \leq \beta \leq b + \rho'(\tau - \tau_0)\}$$

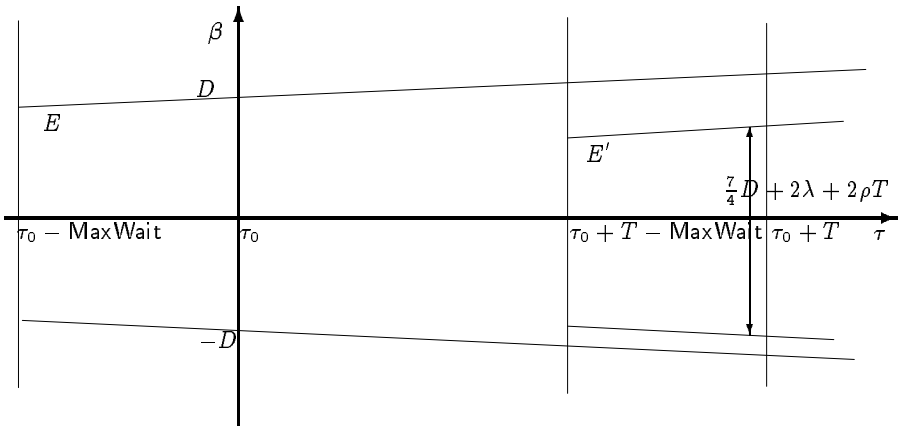
For some real values $a \leq b$ and τ_0. (We also allow $a = -\infty$ or $b = +\infty$).

We sometimes write $E = \text{ENV}\{\tau_0, [a, b]\}$ when we want to stress the parameters τ_0, a and b.

See Figure 3 for examples of a few envelopes. Some notations that we use throughout the proof are summarized below.

- If E is an envelope, we denote by $E(\tau')$ the interval on the β-axis that corresponds to the intersection of E and the line $\tau = \tau'$. For instance, in the notation above $E(\tau_0) = [a, b]$.
  More formally, if $E = \text{ENV}\{\tau_0, [a, b]\}$, then for all $\tau \geq \tau_0$ we have $E(\tau) = [a - \rho(\tau - \tau_0),\ b + \rho(\tau - \tau_0)]$. We also denote by $|E(\tau)|$ the size of the interval $E(\tau)$ (so for example we have $|E(\tau_0)| = b - a$).
- We say that the bias of processor p is in the envelope E during interval $[\tau_1, \tau_2]$, if $B_p(\tau) \in E(\tau)$ for any $\tau \in [\tau_1, \tau_2]$. We say that the bias of p is *not above* E during $[\tau_1, \tau_2]$ if for any $\tau \in [\tau_1, \tau_2]$, we have $B_p(\tau) \leq \max E(\tau)$. Similarly we say that the bias of p is *not bellow* E during $[\tau_1, \tau_2]$ if for any $\tau \in [\tau_1, \tau_2]$, $B_p(\tau) \geq \min E(\tau)$.
- If $E = \text{ENV}\{\tau_0, [a, b]\}$ is an envelope and c is a non-negative number, then we denote by $E + c$ the envelope obtained from E by extending it by c at both sides. That is, $\text{ENV}\{\tau_0, [a, b]\} + c = \text{ENV}\{\tau_0, [a - c, b + c]\}$.

Figure 3: The envelopes E and E' of Lemma 7



- If $E = \text{ENV}\{\tau_0, [a, b]\}$ and $E' = \text{ENV}\{\tau_0, [a', b']\}$ are two envelopes, then the *average* of E and E' is the envelope $avg(E, E') = \text{ENV}\{\tau_0, [(a+a')/2,\ (b+b')/2]\}$. Note that if at some time τ we have one bias value β in E and another bias value β' in E', then the average of these bias values $(\beta+\beta')/2$ is in $avg(E, E')$. (Similarly if we only know that β, β' are not above E, E', respectively, then it follows that their average is not above $avg(E, E')$, and the same for "not below".)

A.2 Main technical lemma

Recall that we denote $T \stackrel{def}{=} (1+\rho)\text{SyncInt} + 2\text{MaxWait}$. Below we assume that $T \leq 2(1-\rho)\text{SyncInt}$. Note that for an interval of length T, the following properties hold:

- In any interval of length T any non-faulty processor performs between one and two SYNCs.
- In any interval of length $T - \text{MaxWait}$ any non faulty processor completes at least one full SYNC. (That is a SYNC that starts and ends within the interval).

We assume that the parameter WayOff is set to $\text{WayOff} = 16\lambda + 18\rho T + \lambda = \delta + \lambda$. Our main technical lemma is as follows.

LEMMA 7. *Let G be a set of at least $n - f$ processors, all of which are non-faulty during some real time interval $[\tau_0 - \text{MaxWait}, \tau_0 + T]$. Also, let $D > 8\lambda$ be a real number. If there exists an envelope E such that $|E(\tau_0)| \leq 2D$, and the biases of G's members are in E throughout the interval $[\tau_0 - \text{MaxWait}, \tau_0]$, then*

i. The biases of G's members remain in E throughout the interval $[\tau_0, \tau_0 + T]$.

ii. Furthermore, there exists an envelope E', $|E'(\tau_0)| = \frac{7D}{4} + 2\lambda$, such that $E' \subseteq E$ and all the biases of all members of G are in the envelope E' in the interval $[\tau_0 + T - \text{MaxWait}, \tau_0 + T]$.

iii. If p is non-faulty in $[\tau_0, \tau_0 + T]$ and $B_p(\tau_0)$ is in $E + \xi$ for some $\xi \geq 0$ then p's bias is in an envelope $E' + \frac{\xi}{2}$ in the interval $[\tau_0 + T - \text{MaxWait}, \tau_0 + T]$.

The envelopes E and E' are depicted in Figure 3. Part (i) of the lemma guarantees that biases of all good members remain within the drift bound. Part (ii) shows that the deviation among the good members actually shrinks (as $\frac{7D}{4} + 2\lambda < 2D$) by the end of the interval T. Finally, part (iii) refers to recovering processors and shows that their distance from the good members halves at each interval T.

This lemma is proved along the lines outlined in Section 4.3. □

A.3 The inductive step

We let $D = 8\lambda + 8\rho T + 2C$. Note that the maximum deviation we prove is $\delta = 2D + 2\rho T$. We look at the time intervals $I_0 = [0, T + \text{MaxWait}]$, and $I_i = [iT - \text{MaxWait}, (i+1)T]$ for $i \geq 1$. For all $i = 0, 1, \ldots$, denote by G_i the set of processors that were non-faulty in the interval of length θ that ends at time $(i+1)T$. (Intuitively, we want to prove that all these processors are synchronized by time $(i+1)T$.) By our assumption on the power of the adversary, we know that the size of each G_i is at least $n - f$. Note also that since $\theta > T + \text{MaxWait}$, then in particular all the processors in G_i are non-faulty during the interval I_i. We prove inductively the following claim:

CLAIM 8. *There are envelopes $E_0, E_1, E_2, \ldots$ such that*

i. $|E_i(iT)| \leq 2D$, and E_i is "almost contained" in E_{i-1}. specifically $E_i \subseteq E_{i-1} + \frac{C}{2}$.

ii. and E_i contains during the interval I_i the biases of all the processors in G_i.

iii. Let $j < i$, and denote $\xi = \max\left(\frac{\text{WayOff}}{2^{i-j}} - \frac{C}{2},\ 0\right)$. Let p be a processor that is non-faulty in $[jT, iT]$. Then, p's bias is in the envelope $E_i + \xi$ during the interval $[iT - \text{MaxWait}, iT]$.

□

Claim 8 immediately implies Theorem 5.