Proc. 17th International Conference on Distributed Computing (DISC 2003) Sorrento, Italy, 121–135. November 13, 2003

Booting Clock Synchronization in Partially Synchronous Systems

Josef Widder*

Technische Universität Wien, Embedded Computing Systems Group E182/2
Treitlstrasse 3, A-1040 Vienna (Austria), widder@ecs.tuwien.ac.at

Abstract. We address the problem of network booting: Distributed processes boot one after the other at unpredictable times in order to start some distributed algorithm; we consider clock synchronization algorithms in systems of $n \geq 3f+1$ processes where at most f exhibit Byzantine behavior. Obviously, assumptions like "there are always at most one third of the running processes Byzantine faulty" do not hold during system startup.
Using a partially synchronous model where upper and lower bounds upon transmission and computation are unknown, we show that a suitable modification of Srikanth & Toueg's non-authenticated clock synchronization algorithm handles network booting and guarantees bounded precision both during normal operation and startup. Accuracy (clocks being within a linear envelope of real-time) is only guaranteed, when sufficiently many correct processes are eventually up and running.

1 Introduction

Synchronized clocks are vital for many applications (see [1] for an overview) and should therefore be provided as early as possible during system operation. Before considering system startup let us first revisit the well-known problem of clock synchronization. Although traditionally studied in systems with known timing behavior where all the processes are equipped with hardware clocks, it can also be solved in partially synchronous systems with software clocks (counters): Every correct process p, which has completed booting and initial synchronization (we call such a process *active*), maintains an integer-valued clock $C_p(t)$, which can be read at arbitrary real-times t. It must satisfy:

(P) **Precision Requirement.** There is some constant *precision* $D_{max} > 0$ such that $|C_p(t) - C_q(t)| \leq D_{max}$ for any two active correct processes p and q and any real-time t.

(A) **Accuracy Requirement.** There are some constants $a, b, c, d > 0$ such that $a(t_2 - t_1) - b \leq C_p(t_2) - C_p(t_1) \leq c(t_2 - t_1) + d$ for any active correct process p and any two real-times $t_2 \geq t_1$.

* This research is part of the W2F-project, which is supported by the Austrian START programme Y41-MAT. (*http://www.auto.tuwien.ac.at/Projects/W2F/*)

According to (P) the difference of any two correct clocks in the system must always be bounded, whereas (A) guarantees some relation between progress of clock-time and progress of real-time. (A) is also called *linear envelope requirement*. The constants a, b, c, and d determine how fast logical time (measured in ticks representing some logical time unit) progresses with respect to real-time. In case of our solution, a clock ticks whenever its clock synchronization algorithm enters the next round of computation. This usually takes one round-trip time, so logical time progresses at about 1 tick per round-trip time.

It is well known [2] that, without authentication[1], no more than one third of the processes may be Byzantine to ensure (P) and (A). Our goal is to handle system startup without further increasing the number of required processes.

Obviously, the startup problem vanishes if an a priori bound on the period of time required for completing the startup of all correct processes can be guaranteed: A process in a (semi-) synchronous system can simply setup a suitable local timeout before it starts sending its first message. However, many real networks cannot be modeled properly as synchronous systems, and even a single correct process that violates the booting time assumption could cause the initialization to fail. Consequently, a time(r)-free initialization, if available, is preferable.

Let us now take a closer look at the problems evolving in systems of $n \geq 3f+1$ processes where all correct processes are initially down. Let us assume that there is an initialization algorithm $\mathcal{I}$ that handles system startup in presence of up to f Byzantine faulty processes. For liveness, $\mathcal{I}$ must of course also handle the case where Byzantine faulty processes are just down and never send messages. Hence it must reach initial synchrony when at least $n - f$ correct processes are up. Now consider the case where Byzantine faulty processes behave exactly as correct ones during initialization whereas f correct ones have not completed booting yet. It is obvious that $\mathcal{I}$ again initializes the system, but now we have $n_{up} \geq 2f + 1$ running processes with f faulty ones among them. Hence, more than one third of the processes would be faulty here.

A straightforward solution could be based upon determining, at runtime, when sufficiently many (at least $2f + 1$) correct processes are eventually up. A process must wait until it has received messages from $3f + 1$ processes for this purpose. To guarantee liveness also when Byzantine processes do not send any messages, however, the number of required processes must be increased to $4f+1$. The question is: Could this penalty somehow be avoided?

Accomplishments: In this paper we show that the number of processes need not be increased for initial clock synchronization in partially synchronous (and hence also synchronous) systems: By modifying the well-known algorithm by Srikanth and Toueg [3], we provide an algorithm that is completely time- and timer-free and requires only $n \geq 3f + 1$ processes even during system startup. It guarantees precision D_{max} during whole system operation, whereas progress of the clocks (accuracy) can only be guaranteed when sufficiently many correct pro-

[1] We do not consider authenticated algorithms since it is never guaranteed that malicious processes cannot break the authentication scheme. Using the algorithm of Srikanth and Toueg [3], our correctness proofs cannot be invalidated by this event.

cesses are up and running. Using our clock synchronization in conjunction with suitable modified higher-level distributed algorithms requiring ε-synchronized clocks may hence lead to solutions which also guarantee higher-level service's safety properties during system startup.

Related Work: Clock synchronization in distributed systems is a very well-researched field, see [2, 4–8] for an overview. Still, there are only a few papers [3, 8–11] known to us that deal with initial synchronization, mostly in the context of integrating a new process in an already running system. For initialization it is often assumed that all correct processes are up and listening to the network when the algorithm is started [3, 9]. In systems where processes boot at unpredictable times, this assumption is too strong. We will see that it can be dropped.

Some solutions for booting exist for very specific architectures: System startup of TTP—viewed as change from asynchronous to synchronous operation—has been investigated in [11]. Initial clock synchronization for the MAFT architecture [10] has been solved, but under stronger system assumptions: A priori assumptions on message transmission delay and local timers are used there to construct a sufficiently large listen window. Termination is achieved by Byzantine Agreement, which, however, requires $2f + 1$ correct processes to be up and running. This cannot always be guaranteed during startup, however. Our goal is minimizing the number of such a priori assumptions. Still we do not know of any approach that could be compared to ours with respect to partial synchrony in conjunction with initially down correct processes.

Our clock synchronization algorithm has *graceful degradation* [12] during system booting. This prevents Byzantine processes from corrupting the system state during the startup phase where more than one third of the running processes may be Byzantine. Mahaney and Schneider [12] introduced synchronous approximate agreement algorithms which provide graceful degradation when between $1/3$ and $2/3$ of the processes are faulty. We reach the same bounds.

If the algorithm would have no graceful degradation, the system could be forced into arbitrary states and the solution for initialization must be *self-stabilizing* [13]. Most self-stabilizing clock synchronization algorithms [14–16] do not stabilize if some processes remain faulty during the whole execution. Exceptions are the algorithms by Dolev and Welch [17], which stabilize even in the presence of Byzantine faults, but they require synchronous systems (enforced by a common pulse) or semi-synchronous systems (processes equipped with physical clocks). This cannot be assumed in partially synchronous models. In fact, self stabilizing algorithms are both unsuitable and an overkill for solving the initial synchronization problem: They cannot guarantee bounded precision during whole system life-time since the transition from illegitimate system states to normal operation cannot always be detected. On the other hand, system startup does not need a stabilizing algorithm's ability to start from arbitrary system state: All processes start from a well-defined state and f must always hold. Self-stabilizing algorithms, however, would be able to recover from a temporal violation of f, at the cost of exponential stabilization times.

Much research has been conducted on partially synchronous systems [18–21]. Clock synchronization is an important issue here as well (see [18, 22]). Our modifications of Srikanth and Toueg's algorithm [3] are in fact based upon ideas from work conducted on consensus algorithms for partially synchronous systems [18, 23]. Still, we do not know of any work that considers system startup. Another system model that is neither completely synchronous nor asynchronous is the Timed Asynchronous Model [24], where processes are equipped with physical clocks. Alternative semi-synchronous models (see e.g. [22]) assume that processes know a priori about the timing bounds of the system. Neither of those requirements is met in our model of partial synchrony.

Our results are related to the crash recovery model [25], where processes crash and recover arbitrarily during the execution of a consensus algorithm. Similar work was conducted in the context of clock synchronization [26]. We, however, consider Byzantine processes and more than $n/2$ "crashed" (actually, "initially dead") processes during startup. This exceeds the bounds used in [25, 26].

Organization of the paper: Section 2 contains the model as well as some notation related to the initialization phase. Our algorithm is described in Section 3. Sections 4 and 5 contain the analysis of the algorithm during startup. In Section 6 we shortly discuss our algorithm's accuracy properties during normal operation. Some analysis had to be omitted due to space restriction and can be found in the full version of this paper [27].

2 System Model

We consider a system of n distributed processes denoted as $p, q, \ldots$, which communicate through a reliable, error-free and fully connected point-to-point network. Even under our perfect communication assumption, messages that reach a process that is not booted are lost. We assume that a non-faulty receiver of a message knows the sender. The communication channels between processes need not provide FIFO transmission, and there is no authentication service.

Among the n processes there is a maximum of f faulty ones. No assumption is made on the behavior of faulty processes; they may exhibit Byzantine faults [28]. Since we investigate network startup, correct processes that just have not booted yet are not counted as faulty.

Our model is partially synchronous [18]. Rather than the global stabilization time model, where it is assumed that the system is synchronous from some unknown point in time on, we use a variant of the model where bounds on transmission and computation delays exist but are unknown. Therefore processes have no timing information and can only make decisions based on received messages.

We employ the partially synchronous model of [29], which is based upon the end-to-end computational + transmission delay δ_{pq} between any two correct processes p and q. It assumes that there are both upper and lower bounds $\tau^+ < \infty$ and $\tau^- > 0$ ensuring $\tau^- \leq \delta_{pq} \leq \tau^+$, which are not known a priori and need not be invariant over time. Note that $\tau^- > 0$ must also capture the case $p = q$; $\tau^+ < \infty$ secures that every message is eventually delivered. The resulting timing uncertainty is measured by the *transmission delay uncertainty* $\varepsilon = \tau^+ - \tau^-$ and the *transmission delay ratio* $\Theta = \tau^+/\tau^-$.

Note that none of τ^-, τ^+, ε, and Θ shows up in our algorithm's code, but only in the formulas established in the analysis of the achieved precision and accuracy. Moreover, it will turn out that the precision D_{max} of our algorithm depends only upon Θ, not upon τ^+ as in classic clock synchronization research. As argued in [30], this has a number of interesting consequences: Since an assumed bound $\overline{\Theta}$ on Θ may still hold when an assumed bound $\overline{\tau}^+$ on τ^+ is violated during periods of overload etc., our algorithm may still work correctly in situations where synchronous ones fail. Actually, τ^+ and τ^- are the worst case and best case response times, respectively, associated with the distributed real-time scheduling problem underlying the execution of our clock synchronization algorithm. Since clock synchronization is a low-level service, typically using high-priority threads and messages, $\varepsilon = \tau^+ - \tau^-$ is usually much smaller than the application-level delay uncertainty $\varepsilon_A = \tau_A^+ - \tau_A^-$. Typical values for Θ reported in real-time systems research [31,32] are $2 \dots 10$. Note also that Θ can be brought down almost arbitrarily close to 1 by introducing additional delays.

2.1 Model of the Initialization Phase

At the beginning all correct processes are down, i.e. they do not send or receive messages. Every message that arrives at a correct process while it is down is lost. A correct process decides independently when it wishes to participate in the system (or is just switched on). Faulty processes may be Byzantine, we can hence safely assume that faulty processes are always up or at least booted before the first correct one. Correct processes go through the following modes:

1. *down*: A process is down when it has not been started yet or has not completed booting.
2. *up*: A process is up if it has completed booting. To get a clean distinction of up and down we assume that a process flushes the input queues of its network interface as first action after booting is completed. Hence it receives messages only if they have arrived when it was up.
   (a) *passive*: Running processes initially perform an initialization algorithm that does not provide the required service to the application. During this phase they are said to be passive.
   (b) *active*: Processes which have completed their initialization in passive mode and provide the required service (in our case clock synchronization) to the application are called active.

Let n_{up} be the number of processes which are up at a given time; n_{up} includes at most f faulty processes.

2.2 Messages

There are only two types of messages sent by our algorithm: *(init, k)* and *(echo, k)*, where k is the sender's clock value. Since processes that started late could have missed previous messages, we require messages with bounded history: The arrival of *(echo, k)* implies the arrival of *(echo, $k-1$)* and *(echo, $k-2$)* messages (this information is transferred implicitly and therefore requires no additional data). Going back two rounds is—as we will see—sufficient for our algorithm.

3 The Algorithm

The algorithm given in Figure 1 is an extension of the classic non-authenticated clock synchronization algorithm by Srikanth and Toueg [3]. The first three **if** clauses (`line 4` to `line 16`) are in fact identical to their algorithm.

```
For each correct process
1  VAR k : integer := 0;
2  VAR mode : {passive, active} := passive;
3
4  if received (init, k) from at least f + 1 distinct processes
5     → send (echo, k) to all [once];
6  fi
7
8  if received (echo, k) from at least f + 1 distinct processes
9     → send (echo, k) to all [once];
10 fi
11
12 if received (echo, k) from at least n − f distinct processes
13    → if mode = active → C := k + 1; fi /* update clock */
14       k := k + 1;
15       send (init, k) to all [once]; /* start next round */
16 fi
17
18 /* catch-up rule */
19 if received (echo, l) from at least f + 1 distinct processes with l > k + 1
20    → if mode = active → C := l − 1; fi /* update clock */
21       k := l − 1; /* jump to new round */
22       send (echo, k) to all [once];
23 fi

Additional Code for each passive correct process
24 if received (init, x) from at least f + 1 distinct processes
25    → C := max(x − 1, k);
26       k := max(x − 1, k);
27       mode := active;
28       send (echo, k) to all [once];
29 fi
```

Fig. 1. Clock Synchronization Algorithm with Startup

The algorithm basically implements a nearly simultaneous global event in a system of $n \geq 3f+1$ processes, which is used to simultaneously increment the clocks at all processes: When a local clock has made its k^{th} tick, the process sends an *(init, k)* message to all. If any correct process receives $f+1$ *(init, k)* messages it can be sure that at least one was sent by a correct process and it therefore sends *(echo, k)*. When $f+1$ *(echo, k)* messages are received at a process it also sends *(echo, k)*. Since correct processes send *(init,k)* and *(echo, k)* only if at least one correct clock has made its k^{th} tick (set $C := k$), we summarize them as messages for the k^{th} tick frequently in our discussion. If a process receives $n-f \geq 2f+1$ *(echo, k)* messages it can be sure that among those are at least $f+1$ messages sent by correct processes. These will be echoed by every other correct process such that, within bounded time, every correct process also receives $n-f$ *(echo, k)* messages – this property is called *relay*. This works because $n-f$ is the minimum number of correct running processes in the system. Therefore, every process that has received $n-f$ *(echo, k)* messages may safely increment its local clock value to $k+1$ and send *(init,* $k+1$*)*.

Srikanth and Toueg showed [3] that the algorithm achieves (P) and (A) for $n \geq 3f+1$ processes if they are initially synchronized. They gave an algorithm for initialization as well, which, however, does not work in our setting: If $n_{up} \geq n-f$ it could be that correct processes make some progress with the "help" of faulty ones. Still, there are not sufficiently many correct processes up to guarantee relay and hence progress at every correct process; (P) and (A) could be violated. A solution for integration of late starters had also been given in [3], which relies on progress which we cannot guarantee for our reduced n_{up}.

We reach our goal of bounded precision (P) during whole system operation based on the following observation: Progress of the clock at any correct process requires always at least $f+1$ messages from distinct correct processes. Since correct processes always send messages to all, every correct process must get those messages. If a process p receives $f+1$ *(echo,* l*)* messages for a future tick l, it can conclude that at least one correct process has the clock value l and process p could update its clock. Therefore we extend Srikanth and Toueg's algorithm by (1) the current clock value k in `line 1` (which is not available in the original algorithm [3] since all ticks are observed concurrently) and (2) the *catch-up rule* in `line 19` that triggers if $f+1$ messages for a future tick are received such that p can update its clock value. Note that a similar construct is used in a clock synchronization algorithm in [18]. In [23] it is used in a phase protocol in a consensus algorithm. Booting is not addressed in those papers, however.

To overcome the problem of lost messages due to initially down processes we add the following protocol (which is not shown in Figure 1): As the first action after getting up, a correct process sends *(echo,* 0*)* to all. If a correct process p receives *(echo,* 0*)* by a process q it resends the last *(echo, k)* message it has sent to q. It is easy to see that the first $f+1$ correct processes are initially synchronized at clock value 0[2], since no correct process can make any progress

[2] When hardware clocks (with different values) must be synchronized initially, resetting all clock values to 0 after booting could be regarded as unsatisfactory. We do

before at least $f+1$ correct processes are up. Before discussing the behavior of correct late starters, we give some useful definitions and lemmas.

Definition 1 (Local Clock Value). *$C_p(t)$ denotes the local clock value of a correct process p at real-time t, and σ_p^k is the real-time when process p sets its local clock to $k+1$, for any $k \geq 0$.*

Definition 2 (Maximum Local Clock Value). *$C_{max}(t)$ denotes the maximum of all local clock values of correct processes that are up at real-time t. Further $\sigma_{first}^k = \sigma_p^k \leq t$ is the real-time when the first correct process p sets its local clock to $k+1 = C_{max}(t)$.*

The following Lemma 1 shows that progress of $C_{max}(t)$ is only possible via the third **if** (`line 12`), which needs at least $f+1$ messages by distinct correct processes. This fact will be heavily used in our proofs.

Lemma 1 (3rd if). *In a system of $n \geq 3f+1$ processes, every correct process—executing the algorithm given in Figure 1—that sets its clock to $C_{max}(t)$ by time t must do so by the third* **if** *clause in* `line 12`.

Proof. By contradiction. Let a correct process p set its clock to $k = C_{max}(t)$ at instant t by a catch-up rule (`line 19` or `line 24`). At least one correct process must have sent a message for a tick $l > k$ before t. Since correct processes never send messages for ticks greater their local clock value, at least one had a clock value $l > k$ at instant t. Thus $C_{max}(t) > k$, which provides the required contradiction. □

Lemma 2 (Minimal Number of Init Messages). *Given an arbitrary point in time t with $l = C_{max}(t)$, let $t' \geq t$ be the instant when C_{max} further increases. For $n \geq 3f+1$, at least $f+1$ correct processes set their clocks to $C_{max}(t)$ by the third* **if** *and therefore send (init, l) before t'.*

Proof. In order for the first correct process to set its clock to $l+1$, it must have received at least $n-f \geq 2f+1$ *(echo, l)* messages sent by distinct processes. At least $f+1$ of those must originate from correct processes, which must have set their local clock to l by the third **if** (Lemma 1) and sent *(init, l)* earlier. □

As we have seen in Lemma 2, $f+1$ *(init, k)* messages are sent for every tick k. Correct processes never send *(init, k)* messages for arbitrarily small ticks compared to C_{max}. These two facts are used for changing from passive to active mode. We have already mentioned that each correct process starts with sending *(echo, 0)* to all. Then it just executes the algorithm from Figure 1. When a correct passive process p eventually receives $f+1$ *(init, x)* messages for a tick x (`line 24`) it can be sure that at least one correct process has sent one. Because x cannot be too far apart from C_{max}, p can conclude that its clock value is within precision and can hence switch to active. We discuss initialization of late starters in Section 5. The precision D_{MCB} of early starters is computed in the following section.

not think so, since, in general, initial hardware clock values are meaningless right after booting if considering internal clock synchronization.

4 Degraded Mode

In this section, we will show that the clocks of all correct early starters are always within D_{MCB} of each other. Note that late starters are guaranteed to eventually reach this precision as well. For now we assume that there is a fixed number $n_{up} \leq n$ of processes which are initially up, and all correct ones (at least $n_{up} - f$) have clock value 0. This models exactly our early phase, starting when enough correct processes are up such that progress is possible but not guaranteed. In this section's analysis we assume that there are no late starters (they are incorporated in Section 5). The following Theorem 1 reveals that even a reduced number of processes achieves certain (weak) properties that are sufficient—as we will see in Theorem 2—to guarantee (P) with some precision D_{MCB}. There is no progress guarantee in this phase and hence no guarantee for (A).

Theorem 1 (Weak Synchronization Properties). *For $n \geq 3f + 1$ with any n_{up}, where $0 \leq n_{up} \leq n$, the algorithm from Figure 1 achieves:*

- *P1W Weak Correctness. If at least $f + 1$ correct processes set their clocks to k by time t, then every correct process sets its clock at least to $k-1$ by time $t+2\tau^+$.*
- *P2 Unforgeability. If no correct process sets its clock to k by time t, then no correct process sets its clock to $k + 1$ by time $t + 2\tau^-$ or earlier.*
- *P3W Weak Relay. If a correct process sets its clock to k at time t, then every correct process sets its clock at least to $k - 2$ by time $t + \varepsilon$.*

Proof. Weak Correctness. If $k = C_{max}(t)$ all $f + 1$ correct processes must have set their clocks to k using the third **if** (`line 12`), and therefore have sent *(init, k)* by time t (see Lemma 1). These correct processes receive the *(init, k)* messages by time $t + \tau^+$ and therefore send *(echo, k)* to all. All correct processes must receive those $f + 1$ *(echo, k)* messages by time $t + 2\tau^+$ and therefore set their clocks to $k - 1$ by the fourth **if** (`line 19`), if they have not already done so.
If $k < C_{max}(t)$ at least $f + 1$ correct processes have set their clocks to k by time $t' < t$ using the third **if** from `line 12` (otherwise no correct process may have set its clock to a value greater than k – see Lemma 1) and therefore all correct processes set their clocks by time $t' + 2\tau^+ < t + 2\tau^+$ (see previous paragraph).

Unforgeability. Setting the clock value can be done via three rules. We derive a contradiction for any of those.
Assume there is a process p that sets its clock to $k + 1$ before instant $t + 2\tau^-$ by the third **if** (`line 12`). It does so because it has received $n - f \geq 2f + 1$ *(echo, k)* messages by distinct processes. There are at least $f + 1$ *(echo, k)* messages sent by correct processes among those, which must have been sent before $t + \tau^-$. Correct processes only send *(echo, k)* when they have received $f + 1$ *(init, k)* or *(echo, l)* messages for some $l \geq k$ from distinct processes. Hence, at least one correct process must have sent a message for tick $l \geq k$ before t. By assumption no tick k messages are sent before t. Since all tick $l > k$ messages are sent after tick k messages by a correct process no such message has been sent by time t, which provides the required contradiction.
Assume that there is a process p that sets its clock to $k+1$ before instant $t+2\tau^-$

using the fourth **if** (`line 19`). Process p does so because it has received $f+1$ *(echo, l)* messages by distinct processes for some $l > k+1$. That is, at least one *(echo, l)* message must have been sent by a correct process q before $t+\tau^-$. Process q has sent it, because it has received at least $f+1$ messages for tick $x \geq l$. At least one of these messages must have been sent by a correct process before time t, since messages for tick $x > k$ are sent by a correct process only after tick k messages. But by assumption P2, no tick k message was sent before t, which again provides the required contradiction.
Assume finally that there is a correct process p that sets its clock to $k+1$ before instant $t+2\tau^-$ using the fifth **if** (`line 24`). Process p does so because it has received at least $f+1$ *(init, $k+2$)* messages by time $t+2\tau^-$. At least one of these *(init, $k+2$)* message must have been sent by a correct process q before $t+\tau^-$. Process q has sent it, because it has received at least $n-f \geq 2f+1$ *(echo, $k+1$)* messages by time $t+\tau^-$, such that at least one (in fact $f+1$) correct process must have sent *(echo, $k+1$)* before time t. A correct process never sends any $k+1$ messages before it has send a message for tick k. By assumption no tick k message was sent by time t which again provides the contradiction.

Weak Relay. Assume $k = C_{max}(t)$. A correct process must set its clock to k using the third **if** from `line 12` (recall Lemma 1), when it has received at least $n-f \geq 2f+1$ *(echo, $k-1$)* messages. Among those are at least $f+1$ messages sent by distinct correct processes. These messages must be received by all correct processes by time $t+\varepsilon$ and therefore they set their clocks to $k-2$ (using the fourth **if**).
If $k < C_{max}(t)$ then at least one correct process has already set its clock to $k' > k$ at time $t' \leq t$ using the third **if** (`line 12`). We have shown in the previous paragraph that all correct processes must set their clocks to $k'-2$ by time $t'+\varepsilon$ so all correct processes must set their clocks to $k-2$ by time $t+\varepsilon$. □

Note that P1W and P3W are directed towards past ticks: Progress is not guaranteed because the properties only ensure that processes reach prior clock values. Still, there are time bounds in P1W, P2 and P3W, which are sufficient to satisfy the precision requirement (P), as we will show in Theorem 2. We require some preliminary lemmas for this purpose.

Lemma 3 (Fastest Progress). *Let p be the first correct process that sets its clock to k at time t. Then no correct process can reach a larger clock value $k' > k$ before $t + 2\tau^-(k'-k)$.*

Proof. By induction on $l = k'-k$. For $l = 1$, Lemma 3 is identical to unforgeability and therefore true. Assume that no correct process has set its clock to $k+l$ before $t+2\tau^- l$ for some l. Thus no correct process may set its clock to $k+l+1$ before $t+2\tau^- l+2\tau^- = t+2\tau^-(l+1)$ by unforgeability. Hence Lemma 3 is true for $l+1$ as well. □

In our analysis we will frequently require to bound the increase of C_{max} during a given real-time interval $[t_1, t_2]$. Lemma 3 can be applied for this purpose if $t_1 = \sigma^k_{first}$ but not for arbitrary times t_1. As in [33] we will provide a general solution based on the following Definition 3.

Definition 3 (Synchrony). *Real-time t is in synchrony with $C_{max}(t)$ iff $t = \sigma^k_{first}$ for some arbitrary k, as defined in Definition 2. Let the indicator function of non-synchrony be defined as*

$$I_{t \neq \sigma} = I_\sigma(t) = \begin{cases} 0 & \text{if } t \text{ is in synchrony with } C_{max}(t), \\ 1 & \text{otherwise.} \end{cases}$$

Lemma 4 (Maximum Increase of C_{max} within Time Interval). *Given any two real-times $t_2 \geq t_1$, $C_{max}(t_2) - C_{max}(t_1) \leq \lfloor \frac{t_2 - t_1}{2\tau^-} \rfloor + I_\sigma(t_1)$.*

Proof. Let $k = C_{max}(t_1) - 1$. We have to distinguish the two cases $\sigma^k_{first} = t_1$ and $\sigma^k_{first} < t_1$. The special case $C_{max}(t_1) = 0$, that is $k = -1$, must be handled as the latter: Since correct processes send *(echo, 0)* when they get up, they are already right in the middle of round 0, thus we must consider $\sigma^{-1}_{first} < t_1$ in this case.

Let $\sigma^k_{first} = t_1$ such that $I_\sigma(t_1) = 0$. From Lemma 3 follows that C_{max} may increase every $2\tau^-$ time-units, hence $\lfloor \frac{t_2 - t_1}{2\tau^-} \rfloor$ times before t_2. Since $I_\sigma(t_1) = 0$, Lemma 4 is true for this case.

Now let $\sigma^k_{first} < t_1$, such that $I_\sigma(t_1) = 1$, and let the real-time $t' = \sigma^{k+1}_{first} > t_1$. We can now apply Lemma 3 starting from time t'. Since $t_2 - t_1 > t_2 - t'$ it follows from Lemma 3 that C_{max} cannot increase more often than $\lfloor \frac{t_2 - t_1}{2\tau^-} \rfloor$ times between t' and t_2. At instant t', C_{max} increases by one such that $C_{max}(t_2) - C_{max}(t_1) \leq \lfloor \frac{t_2 - t_1}{2\tau^-} \rfloor + 1$. Since $I_\sigma(t_1) = 1$ Lemma 4 is true. □

In the following major Theorem 2 we give a bound for the precision requirement (P). We assume an instant t such that $t = \sigma^k_p$ for a correct process p. From weak relay we can derive a bound for σ^{k+2}_q for any other correct process q, such that $\sigma^{k+2}_q = \sigma^{k+2}_{first}$. Using Lemma 4 we can give a bound for $C_{max}(t)$ and hence bound the achievable precision D_{MCB}.

Theorem 2 (Precision in Degraded Mode). *Given a system of $n \geq 3f + 1$ processes with n_{up} processes being up, where $0 \leq n_{up} \leq n$. Let the correct ones among them be initially synchronized to 0. Then the algorithm of Figure 1 satisfies the precision requirement (P) with $D_{MCB} = \lfloor \frac{1}{2}\Theta + \frac{5}{2} \rfloor$.*

Proof. If no correct process advances its clock beyond $k = 2$, precision $D_{MCB} \geq 2$ is automatically maintained since all clocks are initially synchronized to $k = 0$.

Assume that a correct process p has local clock value $k \geq 0$ within a still unknown precision D_{MCB} with respect to all other correct processes—and therefore also to $C_{max}(t')$—at real-time t'. We use weak relay, Definition 3 and Lemma 4 to reason about D_{MCB} by calculating $C_{max}(t)$ for some time $t > t'$.

Let process p advance its clock to $k + 1$ such that $\sigma^k_p = t > t'$. Since p has not done so before t, no other correct process has set its clock to $k + 3$ before $t - \varepsilon$, following directly from weak relay (P3W), thus $\sigma^{k+2}_{first} \geq t - \varepsilon$.

From Lemma 4 follows that $C_{max}(t) \leq \lfloor \frac{t-(t-\varepsilon)}{2\tau^-} \rfloor + I_\sigma(t-\varepsilon) + C_{max}(t-\varepsilon)$. Let us now take a closer look at the term $I_\sigma(t-\varepsilon) + C_{max}(t-\varepsilon)$: If $\sigma^{k+2}_{first} = t - \varepsilon$ and therefore $t-\varepsilon$ is synchronized with C_{max}, $C_{max}(t-\varepsilon)=k+3$ and $I_\sigma(t-\varepsilon)=0$ (following Definition 3). If on the other hand $\sigma^{k+2}_{first} > t-\varepsilon$, $C_{max}(t-\varepsilon)=k+2$ and $I_\sigma(t-\varepsilon)=1$. In both cases $I_\sigma(t-\varepsilon)+C_{max}(t-\varepsilon)=k+3$ such that $C_{max}(t) \le \lfloor \frac{\varepsilon}{2\tau^-} \rfloor + k + 3$ thus $C_{max}(t) \le \lfloor \frac{1}{2}\Theta + \frac{5}{2} \rfloor + k$.

Process p has clock value $C_p(t')=k$ at time $t'<t$ which is by assumption within precision. Since $C_p(t') < C_p(t)$ and $C_{max}(t') \le C_{max}(t)$, we get a bound for D_{MCB} from the difference $C_{max}(t) - C_p(t') = C_{max}(t) - k \le \lfloor \frac{1}{2}\Theta + \frac{5}{2} \rfloor$. □

5 Integration

In the previous section, we discussed the behavior of the early starters. We now turn our attention to correct late starters, which get up after there was possibly some progress of C_{max}. The bound for the resulting precision is derived by a worst case analysis in Theorem 3, where it is assumed that a correct process changes to active mode (and hence must satisfy precision) right after booting based on the oldest possible messages. Due to space restrictions, this section just provides a proof sketch. When considering integration, we must of course guarantee that the required clock synchronization conditions (P) and (A) are eventually satisfied when sufficiently many correct processes got up. This is accomplished in Theorem 4.

Theorem 3 (Precision). *In a system of $n \ge 3f+1$ processes, the algorithm given in Figure 1 satisfies precision (P) with $D_{max} = \lfloor 2\Theta + \frac{11}{2} \rfloor$ throughout whole system operation.*

Proof Sketch. Since *(init)* messages are used for the change from passive to active mode, we must first confirm that no *(init, k)* messages for arbitrarily small k are sent by correct processes, i.e. that $C_{max} - k$ is bounded. This holds true since every correct process requires $n-f$ *(echo)* messages in order to send *(init)*, such that there must be at least one correct process whose message is used to both increase C_{max} and send *(init, k)*.

For the worst case precision, we must consider a newly started process p that receives $f+1$ *(init, k)* messages with k as small as possible compared to C_{max}. Then we give a bound on how much C_{max} can increase before p must change to active at time t, with clock value $k-1$. After time t, p's clock value becomes and stays better, thus $C_{max}(t) - (k-1)$ is our bound for precision D_{max}. □

Theorem 3 shows that (P) is always maintained. In order to show that our algorithm also satisfies (A) when sufficiently many correct processes are up, we give a bound on the maximum time interval it takes to get progress into the system after the $n-f^{th}$ correct processes got up at time t. Since at time t at least one correct process has a clock value of $C_{max}(t)$, *(init, $C_{max}(t)$)* messages may have been missed by the initializing process. The first tick, for which it is guaranteed that all correct processes receive at least $f+1$ *(init)* messages, is $C_{max}(t)+1$. The following Theorem 4 gives the latest possible instant when those *(init, $C_{max}(t)+1$)* messages are received by all correct processes.

Lemma 5. *For the algorithm given in Figure 1 for $n \geq 3f+1$ and $n_{up} \geq n-f$, there are at least $f+1$ correct processes with local clock values $C_{max}(t)$ or $C_{max}(t)-1$ at any time t.*

Proof. If no correct process advances its clock beyond 1 the lemma is true. Let p be some correct process that sets its clock to $C_{max}(t) > 1$ at instant t such that $C_p(t) = C_{max}(t)$. It does so because it has received at least $n-f \geq 2f+1$ *(echo, $C_p(t)-1$)* messages, i.e. at least $f+1$ sent by distinct correct processes. Correct processes never send *(echo)* messages for ticks larger than their local ones. Therefore at least $f+1$ correct processes must have a clock value of $C_p(t)$ or $C_p(t)-1$ at time t. □

Theorem 4 (Initialization Time). *Let t be the time the $n-f^{th}$ correct process p gets up. By time $t+\Delta_{init}$, at least $n-f$ correct processes are running in active mode, where $\Delta_{init} = 8\tau^+$.*

Proof. Process p sends its first *(echo, 0)* message at time t. If there is already progress in the system, p will be initialized quickly because it receives the necessary *(init)* messages τ^+ after they were sent.

If the processes in front are not making progress, the clock value of p at time $t+2\tau^+$ is $C_p(t+2\tau^+) \geq C_{max}(t)-2$ and is reached using the catch-up rule (`line 19`) as the answers from the $f+1$ most advanced processes (recall Lemma 5 and the fact that *(echo, k)* messages have a history, as described in Section 2.2) must be received by then. Process p then sends *(echo, $C_{max}(t)-2$)*, such that by time $t+3\tau^+$ every running correct process must have received $n-f$ *(echo, $C_{max}(t)-2$)* messages.

Every running correct process now sets its clock to $C_{max}(t)-1$ (if it has not yet done so) and sends *(init, $C_{max}(t)-1$)*. Possibly p does not receive $f+1$ *(init, $C_{max}(t)-1$)* messages, because there were too many processes that already had the clock value $C_{max}(t)-1$. So p does not necessarily switch to active mode.

By time $t+5\tau^+$, however, all running correct processes set their clocks to $C_{max}(t)$ and by time $t+7\tau^+$ to $C_{max}(t)+1$. Therefore p and all other running correct processes receive at least $f+1$ *(init, $C_{max}(t)+1$)* messages by time $t+8\tau^+$ or earlier. Then, at least $n-f$ correct processes run in active mode. □

The necessity of $n-f$ messages by distinct processes instead of $2f+1$ is a drawback that cannot be avoided: Assume a system of $n = 5f+2$, thus $4f+2$ correct processes. Assume further that $2f+1$ correct processes start early and reach some arbitrary clock value. Then there could be $2f+1$ correct late starters which could learn about the early starters after they reached a smaller common clock value. During this interval, (P) could be violated.

6 Envelope Synchronization

When $n-f$ correct processes are active, we can give stronger properties than the ones following from P1W, P2 and P3W in Theorem 1. The following envelope condition can be derived from these stronger properties, which also guarantee progress.

Theorem 5 (Accuracy). *The described system, with $n \geq 3f+1$, where at least $n-f$ processes are correct and active, satisfies the following envelope condition*

$$\tfrac{t_2-t_1}{2\tau^+} - 4 + \tfrac{1}{\Theta} < C_p(t_2) - C_p(t_1) < \tfrac{t_2-t_1}{2\tau^-} + D_{max} + 1$$

for every correct active process p and all real-times $t_2 \geq t_1$.

Acknowledgments

I am grateful to Ulrich Schmid, my Ph.D. supervisor, for his support and encouragement.

References

1. Liskov, B.: Practical uses of synchronized clocks in distributed systems. Distributed Computing **6** (1993) 211–219
2. Dolev, D., Halpern, J.Y., Strong, H.R.: On the possibility and impossibility of achieving clock synchronization. Journal of Computer and System Sciences **32** (1986) 230–250
3. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM **34** (1987) 626–645
4. Simons, B., Welch, J.L., Lynch, N.: An overview of clock synchronization. In: Fault-Tolerant Distributed Computing. LNCS 448 (1990) 84–96
5. Ramanathan, P., Shin, K.G., Butler, R.W.: Fault-tolerant clock synchronization in distributed systems. IEEE Computer **23** (1990) 33–42
6. Schneider, F.B.: Understanding protocols for byzantine clock synchronization. Technical Report 87-859, Cornell University, Department of Computer Science (1987)
7. Schmid, U., ed.: Special Issue on The Challenge of Global Time in Large-Scale Distributed Real-Time Systems. Real-Time Systems 12(1–3) (1997)
8. Miner, P.S.: Verification of fault-tolerant clock synchronization systems. NASA Technical Paper 3349 (1993)
9. Lundelius-Welch, J., Lynch, N.A.: A new fault-tolerant algorithm for clock synchronization. Information and Computation **77** (1988) 1–36
10. Kieckhafer, R.M., Walter, C.J., Finn, A.M., Thambidurai, P.M.: The MAFT architecture for distributed fault tolerance. IEEE Transactions on Computers **37** (1988) 398–405
11. Steiner, W., Paulitsch, M.: The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. Proceedings of the The 22nd International Conference on Distributed Computing Systems (2002)
12. Mahaney, S.R., Schneider, F.B.: Inexact agreement: Accuracy, precision, and graceful degradation. In: Proceedings 4th ACM Symposium on Principles of Distributed Computing, Minaki, Canada (1985) 237–249
13. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17** (1974) 643–644
14. Dolev, S., Welch, J.L.: Wait-free clock synchronization. Algorithmica **18** (1997) 486–511
15. Papatriantafilou, M., Tsigas, P.: On self-stabilizing wait-free clock synchronization. Parallel Processing Letters **7** (1997) 321–328

16. Arora, A., Dolev, S., Gouda, M.G.: Maintaining digital clocks in step. Parallel Processing Letters **1** (1991) 11–18
17. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. (1995) Preliminary Version.
18. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323
19. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
20. Attiya, H., Dwork, C., Lynch, N., Stockmeyer, L.: Bounds on the time to reach agreement in the presence of timing uncertainty. Journal of the ACM (JACM) **41** (1994) 122–152
21. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM (JACM) **34** (1987) 77–97
22. Ponzio, S., Strong, R.: Semisynchrony and real time. In: Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92). (1992) 120–135
23. Attiya, H., Dolev, D., Gil, J.: Asynchronous byzantine consensus. Proceedings of the 3rd ACM Symposium of Distributed Computing (1984) 119–133
24. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems **10** (1999) 642–657
25. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. Distributed Computing **13** (2000) 99–125
26. Barak, B., Halevi, S., Herzberg, A., Naor, D.: Clock synchronization with faults and recoveries (extended abstract). In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, ACM Press (2000) 133–142
27. Widder, J.: Switching On: How to boot clock synchronization in partially synchronous systems. Technical Report 183/1-125, Department of Automation, Technische Universität Wien (2002)
28. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4** (1982) 382–401
29. Schmid, U., Fetzer, C.: Randomized asynchronous consensus with imperfect communications. In: 22nd Symposium on Reliable Distributed Systems (SRDS'03). (2003) (to appear).
30. Le Lann, G., Schmid, U.: How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien (2003) (submitted).
31. Ernst, R., Ye, W.: Embedded program timing analysis based on path clustering and architecture classification. In: Digest of Technical Papers of IEEE/ACM International Conference on Computer-Aided Design, IEEE Computer Society (1997) 598–604
32. Gutiérrez, J.P., Garcia, J.G., Harbour, M.G.: Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In: Proceedings of the 10th Euromicro Workshop on Real-Time Systems. (1998) 35–44
33. Schmid, U., Schossmaier, K.: Interval-based clock synchronization. Real-Time Systems **12** (1997) 173–228