HAL
open science

Clock Synchronization in the Byzantine-Recovery Failure Model

Emmanuelle Anceaume, Carole Delporte-Gallet, Hugues Fauconnier, Michel Hurfin, Josef Widder

To cite this version:

Emmanuelle Anceaume, Carole Delporte-Gallet, Hugues Fauconnier, Michel Hurfin, Josef Widder. Clock Synchronization in the Byzantine-Recovery Failure Model. OPODIS, 2007, Pointe-à-Pitre, France. inria-00264219

HAL Id: inria-00264219

https://inria.hal.science/inria-00264219v1

Submitted on 14 Mar 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Clock Synchronization in the Byzantine-Recovery Failure Model

Emmanuelle Anceaume[1], Carole Delporte-Gallet[2], Hugues Fauconnier[2], Michel Hurfin[1], and Josef Widder[3,4,⋆]

[1] IRISA, Campus de Beaulieu, Rennes (France)
[2] LIAFA / Paris VII, Paris (France)
[3] Technische Universität Wien, Vienna (Austria)
[4] École Polytechnique, Palaiseau (France)

Abstract. We consider the problem of synchronizing clocks in synchronous systems prone to transient and dynamic process failures, i.e., we consider systems where all processes may alternate correct and Byzantine behaviors. We propose a clock synchronization algorithm based on periodical resynchronizations which is based on the assumption that no more than $f < n/3$ processes (with n the number of processors in the system) are simultaneously faulty. Both, accuracy (clocks being within a linear envelope of real-time) and precision (maximum deviation between clocks) perpetually hold for processes which sufficiently long follow their algorithm. We provide expressions for both the recovery time and the failure turn-over rates. Both expressions are independent of f, and are less than the time needed to execute 3 resynchronizations.

1 Introduction

Tightly synchronized and accurate clocks among the members of a distributed system is a fundamental service as it allows, e.g., to perform synchronized actions or estimate the behavior of the environment in a control system. One way to ensure reliable and tight synchronization among local clocks is the use of a clock synchronization algorithm. Essentially such an algorithm overcomes clock drift, variations of transmission delays and failures. It guarantees that the maximum deviation between (correct) local clocks is bounded (precision) and that these clocks are within a linear envelope of real-time (accuracy).

There is considerable literature devoted to the design and implementation of clock synchronization algorithms; see [1–5] for an overview. Some algorithms are specified for environments in which processes may crash [6], may suffer timing failures [7], or may execute arbitrarily bad operations [1, 8–10]. The last type of behavior, called Byzantine, is the most severe type of process failures. It captures all causes of failures, ranging from accidental memory bit flips to malicious attacks on a system. Therefore this model seems appropriate for a large range of distributed applications.

Another kind of fault tolerance is self-stabilization. Here it is assumed that the system behaves arbitrarily (including, e.g., that the assumed threshold of faults is temporarily violated) but if eventually all processes behave according to the algorithm the system

⋆ Partially supported by the Austrian FWF project *Theta* (proj. no. P17757).

stabilizes to a good state (where in our case the clock synchronization properties hold). A highly interesting work is with respect to joining the two approaches: Self-stabilizing clock synchronization algorithms that work in the presence of permanent Byzantine faults are given in [11, 12]. However, these solutions share some properties which seem inherent to the problem of fault-tolerant self-stabilization: First, even processes that always follow their algorithm are not guaranteed to remain synchronized to each other (this is clearly due to well known bounds on resilience [1] which are violated during unstable periods) and second, resynchronization of recovered processes takes $O(f)$ time.

This paper is based on the idea, that permanent failures are too optimistic for certain applications, while fault-tolerant self-stabilization might be too pessimistic, or the provided properties too weak. We therefore explore under which conditions clock properties can be provided permanently in the presence of transient and dynamic Byzantine faults, where processes recover from "bad periods" with an arbitrary state and just start following their algorithm. We, however, limit the number of components which may suffer from faults simultaneously.

This Byzantine-recovery failure model has been previously investigated in [13, 14] (both work will be discussed in Section 6). In [14], the work is motivated by security schemes for which a clock synchronization algorithm under this failure model is more robust than others. However, our motivation for this failure model comes from long-lived applications in the space domain. There, transient and repeated bit-flips phenomena, caused by single event upsets (SEU), may impact processors of the computing system. In addition, mission times can be extremely long, rendering unrealistic the hypothesis that there is a limitation on the number of faults that may occur during the application life, and that only a subset of the processors can be affected by these faults. To deal with such strong requirements, complex checking procedures are designed, and reconfiguration and/or correcting mechanisms are applied on the altered components. Such mechanisms ensure that altered processors recover some operational state, mainly they recover a correct execution code. Clearly, recovering an operational state does not mean recovering a safe state, i.e., having the clock synchronized, for example. To summarize, the notion of faulty and correct processors does not make sense in the Byzantine-recovery failure model, "correct" in the sense that a processor is correct for the whole mission. Rather, processors alternate between periods of time during which they are faulty, and periods of time during which they follow their prescribed protocol.

Contribution. We propose a clock synchronization algorithm tolerant to moving Byzantine failures. In particular our algorithm guarantees that in presence of up to f "moving" and concurrent Byzantine failures, correctly behaving processes (that is at least $n - f$ processes, with $n \geq 3f + 1$, and n the number of processors in the system) have synchronized logical clocks. Our algorithm is a variation of Srikanth and Toueg's clock synchronization algorithm [9], in which the classic notion of "correct process" is assumed. The challenge of the present work is the guarantee that correctly behaving processes are never corrupted by recovering processes, and that clocks of recovering processes get quickly tightly synchronized with those of correctly behaving processes. We provide an expression for the recovery time (i.e., the period of time after which a recovered process is synchronized with the other processes). This bound is independent

of f, and is roughly equal to the time needed to execute two resynchronizations. We derive an expression for the failure turn-over rate, i.e., the maximal allowable frequency at which processes may enter (and leave) faulty periods. This rate is also independent of f, and is roughly equal to the time needed to execute three resynchronizations.

2 System Model and Problem Statement

Network and Clock Model. The system consists of a finite set of $n \geq 3f + 1$ processes, where f is a parameter which is used below for our failure assumption. Processes communicate and synchronize with each other by sending and receiving messages over a (logically) fully connected reliable point-to-point network. The system is synchronous, as that there exists known upper and lower bounds on processing speeds; every process has access to a hardware clock with bounded drift with respect to Newtonian real-time; and there is a known upper bound on messages transmission delays. More precisely, we assume the following:

1. The rate of drift of physical clocks from real-time is bounded by a known constant $\varrho > 0$. That is, if $H_p(t)$ is the reading of the hardware clock of process p at real-time t, then for all $t_2 \geq t_1$:

$$\frac{t_2 - t_1}{1 + \varrho} \leq H_p(t_2) - H_p(t_1) \leq (1 + \varrho)(t_2 - t_1)$$

   The rate of drift between clocks is consequently bounded by $dr = \varrho \cdot \frac{2+\varrho}{1+\varrho}$.
2. There is an upper bound δ on the time required for a message to be prepared by a process, sent to a process and processed by the recipient of the message.

Failure Model. As written above, we want to model transient and dynamic process failures, i.e., processes may temporarily (permanent process failures are just a special case) deviate from the specified behavior. For example, such a process may arbitrarily change its local state, omit to send messages, may change the content of its messages or may even generate spurious messages. Note however, that we exclude masquerading by our logical point-to-point assumption. Further we want to model recovery such that process p reaches an operational state whenever p recovers a correct code, and makes steady progress in its computation, i.e., follows its algorithm. Note that p's execution context may be still altered (similar to self-stabilization), and thus p may still be perceived as faulty as long as it has not reached a safe state, i.e., an internal state that satisfies problem specific invariants (e.g., having its logical clock synchronized). The time needed to reach a safe state from an operational one is called the *recovery time*, and is denoted in the following by j.

Definition 1 (Obedient Processes). *We denote by Obedient*(t_1, t_2) *the set of processes that follow their algorithm during the whole real-time interval* $[t_1, t_2]$*, and by* $\mathcal{P}_\Delta(t)$ *the set in Obedient*$(max\{0, t - \Delta\}, t)$*, with* Δ *being some constant real-time interval.*

Definition 2 (Fault Model). *For every real-time $t > 0$ it holds that*

$$|\mathcal{P}_m(t)| \geq n - f \tag{1}$$

with m being some constant real-time (the fault turn-over interval), and $n \geq 3f + 1$. Initially, at time $t = 0$, all processes are in an initial (i.e., correct) state.

This definition states that in a sliding window of length m, the number of processes that can concurrently exhibit faulty behavior is no more than f, with $n \geq 3f + 1$. With $m = \infty$ there are at least $n - f$ "correct" processes, while at most f may fail during an execution: We get similar restrictions as in the classic Byzantine fault model [15, 16].

Problem Statement. As previously said, a clock synchronization algorithm allows processes to update their local clocks to overcome drifts and failures. Process p's local clock (also called in the literature p's logical clock) at real-time t, denoted $C_p(t)$, follows its hardware clock $H_p(t)$ with periodic re-adjustment. A Δ-Clock Synchronization algorithm has to satisfy the following two properties:

(π) *Precision.* At any real-time $t \geq 0$ and for any two processes $p, q \in \mathcal{P}_\Delta(t)$ it holds for some constant $D_{\max}$ that

$$|C_p(t) - C_q(t)| \leq D_{\max}$$

(α) *Accuracy.* For any process p and for any two real-times s and e with $p \in$ *Obedient*$(s, e) \wedge (e - s) > \Delta$ it must hold for any two real-times $t_1, t_2 \in [s + \Delta, e]$, $t_1 < t_2$, for some constants a, b, c, and d that

$$\frac{t_2 - t_1}{a} - b \leq C_p(t_2) - C_p(t_1) \leq (t_2 - t_1)\, c + d$$

Precision ensures that the maximum deviation between logical clocks of any two processes that are obedient for at least Δ real-time units is bounded. Accuracy guarantees that the logical clock of a process obedient for at least Δ real-time units remains in the linear envelope of real-time.

3 The Algorithm

Algorithm 1 is a variant of the non-authentication clock synchronization algorithm by Srikanth and Toueg [9]. Its rules (starting with "*on*") are executed atomically. There are several data structures, namely *Buffer* and *timestamps*, where *Buffer*$_p[q]$ contains the last resynchronization message sent by q that p received, and *timestamps*$_p[q]$, p's local time at which p received that resynchronization message. The algorithm relies on several parameters. The (local) interval P between two executions of the resynchronization protocol, the delete interval parameter R which is the time interval during which resynchronization messages are locally kept within *Buffer*, and the adjustment parameter A guaranteeing that logical clocks of processes which are obedient for sufficiently

long are not set back. All these parameters are computed from the estimation of system parameters δ and ϱ. They have to satisfy the following solvable set of constraints, constraints that will be discussed in the remainder:[5]

$$A \geq r \cdot (1+\varrho) \qquad P > (3 \cdot \delta) \cdot (1+\varrho) + A + R \cdot (1+\varrho)$$
$$R = r \cdot (1+\varrho) \qquad r = (P - A) \cdot dr + 3 \cdot \delta$$

After discussing the general principles of our algorithm, we will show that it solves Δ-Clock Synchronization under the failure assumption of Definition 2 for $\Delta = j$ and m as follows (with an infinitesimally small ε):

$$j \geq 2 \cdot r + P \cdot (1+\varrho) \qquad m \geq j + R \cdot (1+\varrho) + \delta + \varepsilon$$

Srikanth and Toueg's Algorithm [9]. We briefly discuss the principles of their algorithm. It is based on processes which are either "correct" or "faulty" permanently. The resynchronization proceeds in rounds, a period of time during which processes exchange messages and update their logical clocks: When the logical clock of some correct process shows time $k \cdot P$, with $k \geq 1$, this process sends a message to all, indicating that it is ready to resynchronize. When a correct process receives $f+1$ such messages, it knows that at least one was sent by a correct process, and thus that at least one correct process is ready to resynchronize. Therefore it also sends such a message to all. Upon receipt of a resynchronization message from $n - f \geq 2f + 1$ processes, process p knows that all correct processes will receive at least $n - 2f \geq f + 1$ of these messages within bounded time, and will therefore send their resynchronization messages to all, such that in turn every correct process receives $n - f$ such messages within bounded time. Thus, p "accepts" this message and resynchronizes its logical clock to $k \cdot P + A$.

Our Algorithm. Intuitively, the main problem in the dynamic fault model is that a process has to get rid of messages which it receives from a, then faulty, process for "future" rounds, i.e., for too large values of k. In the static failure model this is simpler to overcome since such messages are sent just by the at most f faulty processes during the whole execution, while in the dynamic model such messages may be sent by every process at times it does not follow its algorithm.

The structure of our algorithm is similar to [9]. Resynchronizations are triggered periodically (line `7`), and if properly relayed, and agreed by sufficiently many processes resynchronization is applied by all the processes in $\mathcal{P}_j(t)$ (line `25`). To prevent too much bad information from being present in *Buffer*, invalid messages are deleted from *Buffer* (line `11`). A message is invalid if it belongs to *Buffer* for more than R logical time units, or if its reception time is in the future; R corresponds to the maximal time needed to properly complete a resynchronization phase. To prevent incorrect resynchronizations,

[5] The constraints are given here as required for the proofs. At first sight there seem to be cyclic dependencies. However, by simple arithmetical manipulation one can derive that A does in fact only depend on δ and ϱ while P must be greater than A plus a term depending again on δ and ϱ. The system is sound if $\varrho < 0.32$, which is in practice given as hardware clocks have drift rates between 10^{-6} and 10^{-4}.

Algorithm 1 Clock Synchronization Algorithm

```
1:  variables
2:      k ← 1                                                          // round number
3:      vector of integers: Buffer[1 . . . n] ← ⊥
4:      vector of real: timestamps[1 . . . n] ← 0
5:      real: C(t) ← 0
6:      sent ∈ {TRUE, FALSE} ← FALSE

7:  on C(t) = k · P do
8:      if sent = FALSE then
9:          send (TICK, k) to all
10:     sent ← TRUE

11: on ((timestamps[q] < C(t) − R) ∨ (C(t) < timestamps[q])) ∧ (Buffer[q] ≠ ⊥) do
12:     Buffer[q] ← ⊥

13: on receipt of message (TICK, ℓ) sent by process q do
14:     Buffer[q] ← ℓ
15:     timestamps[q] = C(t)
16:     if |{r : Buffer[r] = ℓ}| ≥ f + 1 ∧ ℓ = k then
17:         if sent = FALSE then
18:             send (TICK, ℓ) to all
19:         sent ← TRUE
20:     if |{r : Buffer[r] = ℓ}| ≥ n − f then
21:         for all r ∈ Π do
22:             timestamps[r] ← timestamps[r] + (ℓ · P + A − C(t))
23:             if Buffer[r] = ℓ then
24:                 Buffer[r] ← ⊥
25:         C(t) ← ℓ · P + A
26:         k ← ℓ + 1                                                  // set round number
27:         sent ← FALSE
```

process p relays a (TICK, k) message only if it makes sense for itself, i.e., (1) p is sure that at least one process with a "good" internal state wants to resynchronize, (2) both q and p agree on the resynchronization round (line 16), and (3) p has not already sent (TICK, k).

The presence of $n-f$ (TICK, k) messages in p's buffer is sufficient to resynchronize its clock, i.e., to set it to $k \cdot P + A$. This also allows a recovering process p to resynchronize its clock. Note that this resynchronization need not be based on "real" messages, as p may still have bad information in its buffer that is due to the time when it did not follow its algorithm, and thus it may synchronize to a wrong clock. However, the algorithm guarantees that at the end of the next resynchronization, p will have cleaned its buffer, and will be able to resynchronize its clock with all the other correct processes. We did not explicitly handle overruns of the round number k or the clock variables. With assumptions on the mission duration, variables can be dimensioned sufficiently large, such that overruns only happen due to local faults such that variables can be reset safely.

4 Properties of the Algorithm

In this section we give the main lines of the correctness proofs. (See [17] for the complete proofs.) We begin by defining some intensively used notations.

If some process $p \in \mathcal{P}_j(t)$ takes some step s at real-time t then we say that p *properly executes* s and if some process $p \in \mathcal{P}_j(t)$ sends some message m at real-time

t we say that p *properly sends* m. If p properly executes line 26, then p *terminates* round ℓ. Moreover, we will heavily use t_{del} which is defined to be 2δ. We now give some preliminary definitions, the first of which are similar to [9].

Definition 3. *For each round k, the instant the first process properly sends (TICK, k) is denoted by* ***ready^k****. The time the $f+1^{st}$ process properly sends (TICK, k) is denoted* ***go^k****. The time the $n-f^{th}$ process properly sends (TICK, k) is called* ***fast-go^k****. Finally, the time the first (or last) process properly terminates round k and sets its clock to $kP+A$ is denoted* ***beg^k*** *(or* ***end^k****, resp.). Further, based on these times we define:*

$$\mathcal{O}_k = \mathit{Obedient}(\mathit{ready}^k, \mathit{fast\text{-}go}^k + \delta)$$
$$\mathcal{C}_k = \mathit{Obedient}(\mathit{ready}^k - r, \mathit{fast\text{-}go}^k + \delta)$$
$$\mathcal{S}_k = \mathit{Obedient}(\mathit{ready}^k - j, \mathit{fast\text{-}go}^k + \delta)$$

The following properties form a central part of the analysis. Essentially these properties impose constraints on local structures at the start of a resynchronization period. In this section, we assume initial synchronization, i.e., we assume that the properties hold for $k=1$. A large part of the analysis is then devoted to prove that the properties are in fact invariants of the algorithm. On these invariants we later build our proofs for (π) and (α). We discuss in Section 5 how and under which assumptions initial synchronization can be achieved.

Invariant 1 *With respect to round $k \geq 1$, we define:*

(S) Synchronized Start. *fast-go^k − ready^k $\leq t_{\mathrm{del}} + (P-A)\cdot dr$.*
(N) Consistent round numbers. *At ready^k − ε (for an infinitesimally small ε) all processes in $\mathcal{P}_j(\mathit{ready}^k)$ have a round number equal to k and sent* = FALSE.
(B) Clean buffer. *At real-time ready^k − ε all processes in $\mathcal{P}_j(\mathit{ready}^k)$ have Buffer$[p] = \perp$ and there are no messages in transit on outgoing links of processes $p \in \mathcal{P}_j(\mathit{ready}^k)$.*
(C) $|\mathcal{S}_k| \geq n-f$.
(T) *No process has properly sent a message (TICK, ℓ) for $\ell \geq k$ before ready^k.*

In the case of classic, i.e., static Byzantine faults, a consequence of the model is, that at no time, a correct process has received messages by more than f faulty processes. In our case, we neither have the notion of a correct nor of a faulty process. In order to achieve clock synchronization — and thus to circumvent the lower bound in [1] — we have to ensure that not too much bad information is present at processes which should ensure properties (π) and (α).

Recall that the fault turn-over interval m (see Definition 2) satisfies the following relation: $m \geq j + R\cdot(1+\varrho) + \delta + \varepsilon$, with an infinitesimally small ε, and the recovery time j is such that $j = 2\cdot r + P\cdot(1+\varrho)$. Intuitively, m must be greater than the recovery time j (otherwise the adversary could corrupt all the processes in the system by moving fast enough from one to another one), and must face situations in which some process p that recovered a safe state at time t (i.e., p enters $\mathcal{P}_j(t)$) may have sent "wrong" messages right before t. Thus buffers have to be cleaned (which takes $\delta + R\cdot(1+\varrho) + \varepsilon$ real time units) before the adversary is allowed to break into new processes. Then we have:

Lemma 1 (Clean State). *At all times t, any process $p \in \mathcal{P}_j(t)$ has less than or equal to f values different from $\perp$ in the vector Buffer which were not received via properly sent messages.*

Proof. Suppose by way of contradiction that p has more than f values in his vector *Buffer* which it wrote in line `14` due to a message sent by some process q at some time $t' \leq t$ such that $q \notin \mathcal{P}_j(t')$. By line `11`, no values which are older than R are kept in p's vector *Buffer*. Thus messages by more than f distinct processes must have been sent at some times t' such that these processes where not in $\mathcal{P}_j(t')$ and $t - R \cdot (1 + \varrho) - \delta \leq t' \leq t$.

As $|\mathcal{P}_m(t)| \geq n - f$ and $m \geq j + R \cdot (1 + \varrho) + \delta + \varepsilon$ it follows that $\mathcal{P}_j(t') \supseteq \mathcal{P}_m(t)$. Consequently, $|\bigcup_{t'} p \notin \mathcal{P}_j(t')| \leq f$ which provides the required contradiction to p having received messages by more than f distinct processes. □

We now investigate properties of the round structure. We show two basic properties of our algorithm, which are named after similar properties of the broadcasting primitive in [9, 18], i.e., unforgeability and correctness.

Lemma 2 (Unforgeability). *If a process properly terminates round k at time t, then at least one process properly has sent (TICK, k) at some time $t' \in [t - R \cdot (1 + \varrho) - \delta, t]$.*

Proof. Assume by contradiction that $q \in \mathcal{P}_j(t')$ terminates round k at time t', although no message (TICK, k) was properly sent in the given interval. Due to line `20`, it does so because it has at least $n - f \geq f + 1$ entries in *Buffer* for round k. By Lemma 1 no more than f of these are due to processes not in $\mathcal{P}_j(t'')$ when they send (TICK, k) at time t'', with $t'' \leq t'$. Thus at least one process must have properly sent (TICK, k) within the interval (otherwise it would have been deleted by time t in line `11`) which provides the required contradiction. □

Lemma 3. *The first process that properly sends (TICK, k) does so in line `9`.*

Lemma 4. *No process properly terminates round k at some time $t' < go^k$.*

Lemma 5. *For every k, if (S) then (C).*

Proof. By Definition 2, $|\mathcal{P}_m(t)| \geq n - f$, for all t. Consequently, it suffices to show that $m \geq \textit{fast-go}^k + \delta - \textit{ready}^k + j$. By (S), $\textit{fast-go}^k + \delta - \textit{ready}^k + j \leq \delta + (P - A)\, dr + t_{\text{del}} + j = r + j$. Further, $m = j + R(1 + \varrho) + \delta + \varepsilon$ such that it follows that $m > \textit{fast-go}^k + t_{\text{del}} - \textit{ready}^k + j$ which concludes the proof. □

We now present some lemmas which are all built upon properties (S), (N), (B), (C), and (T) . These lemmas are used in the induction proof of Theorem 2.

Lemma 6. *Suppose (S), (N), (B), (C), and (T) hold for round k. Then process $p \in \mathcal{O}_k$ does not remove any messages that are sent within $[\textit{ready}^k, \textit{fast-go}^k + \delta]$ within this interval via line `11`.*

Proof. Only messages older than R on p's logical clock are removed (in line `11`). The minimum real-time duration for R is $\frac{R}{1+\varrho}$ which is r. Consequently only messages sent before $\textit{fast-go}^k + \delta - r$ are removed. By (S), $\textit{fast-go}^k + \delta - r < \textit{ready}^k$ which concludes the proof. □

Lemma 7. *Suppose (S), (N), (B), and (C), and (T) hold for round k. Then no process in $\mathcal{S}_k$ sends a (TICK, ℓ), with $\ell \neq k$, message within $[\textit{ready}^k, \textit{fast-go}^k + \delta]$.*

Lemma 8. *Suppose (S), (N), (B), (C), and (T) hold for round k. Then let some process $p \in \mathcal{O}_k$ receive (TICK, k) messages by at least $n - f$ distinct processes in $\mathcal{S}_k$ within $[\textit{ready}^k, t]$, with $\textit{ready}^k \leq t \leq \textit{fast-go}^k + \delta$.*

1. *p terminates round k within $[\textit{ready}^k, t]$.*
2. *After terminating round k within $[\textit{ready}^k, t]$, p does not terminate round ℓ for some $\ell \neq k$ by $go^k + t_{\text{del}}$.*

Lemma 9 (Correctness). *Suppose (S), (N), (B), (C), and (T) hold for round k. Then every process in $\mathcal{O}_k$ terminates round k within $[\textit{ready}^k, go^k + t_{\text{del}}]$.*

Proof. By go^k, $f + 1$ processes properly send (TICK, k). These messages are received by all processes in $\mathcal{S}_k$ (which have a clean *Buffer* due to Lemma 1) such that by time $go^k + \delta$ at least $f + 1$ messages are in their buffer. These processes send (TICK, k) by time $\textit{fast-go}^k \leq go^k + \delta$ due to line `18`. Thus the messages by these at least $n - f$ distinct processes are received by all processes in $\mathcal{O}_k$ within $[\textit{ready}^k, \textit{fast-go}^k + \delta]$. By Lemma 8, our lemma follows. □

Lemma 10. *Suppose (S), (N), (B), (C), and (T) hold for round k. Then:*

1. *Every $p \in \mathcal{C}_k$ terminates round k exactly once within $[go^k, go^k + t_{\text{del}}]$.*
2. *At time $go^k + t_{\text{del}}$, p has at most f messages for round k sent by processes in $\mathcal{S}_k$ and at most f messages which where not sent properly in Buffer.*

Proof. As processes in $\mathcal{C}_k$ follow their algorithm at least r before $\textit{ready}^k$, they have deleted all messages they had in their *Buffer* that were due to a time where they possibly did not follow their algorithm in line `11`.

Due to Lemma 9 and by similar reasoning with which one can show Lemma 4, every process $p \in \mathcal{C}_k$ terminates round k at some time $t \in [go^k, go^k + t_{\text{del}}]$ at least once. To prove (1), let p do so such that it removes all messages from *Buffer* for round k in line `24`. It does so based on $n - f$ received messages, i.e., at least $n - 2f$ messages by processes in $\mathcal{S}_k$. Only one[6] (TICK, k) message sent by each process in $\mathcal{S}_k$ is received such that no more than f messages from processes in $\mathcal{S}_k$ can be received after t. Consequently, p cannot reach the $n - f > 2f$ threshold necessary to execute line `26` (and terminate round k) within $[t, go^k + t_{\text{del}}]$.

The first part of (2) is a consequence of the proof of (1), while the second part of the proof is identical to the proof of Lemma 1. □

Let in the remainder of this section e^k be the time the last process in $\mathcal{C}_k$ terminates round k within $[go^k, go^k + t_{\text{del}}]$ and also fix the real-time $\tau = e^k + (P - A)(1 + \varrho)$.

Lemma 11. *For every process $p \in \mathcal{P}_j(\tau)$ it holds that $p \in \mathcal{C}_k$.*

[6] Processes follow their algorithm, i.e., cannot terminate a round other than k within the time window, consequently they cannot set their round number to k (and set *sent* to FALSE) which would be required to re-send a message.

Proof. First we have to show that $\tau - j \leq \mathit{ready}^k - r$. According to its definition, $j = 2 \cdot r + P \cdot (1 + \varrho)$. We have to show that $\tau - \mathit{ready}^k \leq r + P \cdot (1 + \varrho)$, i.e.,

$$e^k - \mathit{ready}^k \leq (P - A) \cdot dr + \delta + t_{\mathrm{del}} + A(1 + \varrho) \tag{2}$$

By property (S), $\mathit{fast\text{-}go}^k - \mathit{ready}^k \leq t_{\mathrm{del}} + (P - A) \cdot dr$ and by Lemma 8, $e^k \leq \mathit{fast\text{-}go}^k + \delta$. Consequently, we know that $e^k - \mathit{ready}^k \leq 3\delta + (P - A) \cdot dr$ which — by the size of A — proves Equation (2).

Second we have to show that $\mathit{fast\text{-}go}^k + \delta \leq \tau$, i.e., that

$$\mathit{fast\text{-}go}^k + \delta \leq e^k + (P - A)(1 + \varrho). \tag{3}$$

Since by definition of P, $P - A > (3 \cdot \delta) \cdot (1 + \varrho) + R(1 + \varrho)$, we can prove Equation (3) by showing that $\mathit{fast\text{-}go}^k \leq e^k + 2\delta$.

By Lemma 4, $\mathit{go}^k \leq \mathit{beg}^k$. Since by time $\mathit{go}^k + \delta$ all processes in $\mathcal{S}_k$ receive $f + 1$ (TICK, k) messages and therefore send (TICK, k) it follows that $\mathit{fast\text{-}go}^k \leq \mathit{go}^k + \delta$ since $|\mathcal{S}_k| \geq n - f$. As $\mathit{go}^k \leq \mathit{beg}^k \leq e^k$ it follows that $\mathit{fast\text{-}go}^k \leq e^k + \delta$ and thus our lemma follows. □

Lemma 12. *If (S), (N), (B), (C), and (T) hold for round k, then no messages are properly sent within $[\mathit{beg}^k + t_{\mathrm{del}}, \mathit{ready}^{k+1}]$, for any $k > 0$.*

Proof. Lemma 11 in conjunction with Lemma 10 shows that all processes $p \in \mathcal{P}_j(\tau)$ update their round number once to $k + 1$ within t_{del}. By Lemma 10(2), there are not sufficiently many messages in transit such that p can execute line `18` before the first process in $\mathcal{C}_k$ has properly sent $(\mathrm{TICK}, k + 1)$, while there are also not sufficiently many messages (i.e., less than $n - f$) to execute line `25` before the first process in $\mathcal{C}_k$ has sent a message. Thus processes properly execute no rule (except line `11`) before the first clock of a process in $\mathcal{C}_k$ properly reaches $(k + 1) \cdot P$ which is not before $\mathit{beg}^k + \frac{P-A}{1+\varrho}$ which thus is a lower bound for ready^{k+1}. □

Lemma 13 (Monotony). *Suppose (S), (N), (B), (C), and (T) hold for round k. If $p \in \mathcal{P}_j(\tau)$ terminates round k within $[\mathit{ready}^k, \mathit{beg}^k + t_{\mathrm{del}}]$, then at no time t, $\mathit{beg}^k + t_{\mathrm{del}} < t \leq \tau$, p terminates round k.*

Proof. Suppose $p \in \mathcal{P}_j(\tau)$ terminates round k within $[\mathit{ready}^k, \mathit{beg}^k + t_{\mathrm{del}}]$. By Lemma 12, from $\mathit{beg}^k + t_{\mathrm{del}}$ on, no process properly sends (TICK, k). Within $[\mathit{ready}^k, \mathit{beg}^k + t_{\mathrm{del}}]$, no process properly sends (TICK, ℓ), with $\ell < k$ (Lemma 7). By an argument similar to the one used for Lemma 2, the lemma follows. □

After all these preliminary lemmas, we finally arrive at our major theorem. If initial synchronization is given one may set $\sigma = 0$. For our initialization algorithm, however, σ will be 2.

Theorem 2. *Algorithm 1 ensures that for all $k \geq \sigma$ the properties (S), (N), (B), (C), and (T) as well as $e^k - \mathit{beg}^k \leq t_{\mathrm{del}}$ are satisfied given that the properties (S), (N), (B), (C), and (T) hold for some round $\sigma \geq 0$.*

Proof. The proof is by induction on k. For $k = \sigma$, (S), (N), (B), (C), and (T) hold since the properties of initial synchronization are assumed to hold. By Lemma 10 the base case follows.

Now assume that (S), (N), (B), (C), and (T) hold for all ℓ, with $\sigma \leq \ell < k+1$ and $e^\ell - \mathit{beg}^\ell \leq t_{\mathrm{del}}$. We have to show that they hold for round $k+1$. Relation $e^{k+1} - \mathit{beg}^{k+1} \leq t_{\mathrm{del}}$ then follows directly from Lemma 10.

(S) All processes $p \in \mathcal{P}_j(\tau)$ send $(\textsc{tick}, k+1)$ at the latest by $e^k + (P - A)(1+\varrho)$ which constitutes an upper bound for $\mathit{fast\text{-}go}^{k+1}$. From the induction assumptions and Lemma 10 it follows that $e^k - \mathit{beg}^k \leq t_{\mathrm{del}}$. Thus $\mathit{fast\text{-}go}^{k+1} - \mathit{ready}^{k+1} \leq t_{\mathrm{del}} + (P - A)\,\mathit{dr}$ which proves (S) for $k+1$.

(N) Since no rules (except line `11`) are executed after processes in $\mathcal{P}_j(\tau)$ have set their round number to $k+1$, their round number remains unchanged and $\mathit{sent} = \textsc{false}$ as it is set to this value when the round number is updated.

(B) As no rules (except line `11`) are properly executed by processes in $\mathcal{C}_k$ between e^k and ready^{k+1}, no messages are sent by processes in $\mathcal{P}_j(\mathit{ready}^{k+1})$ in this interval, and all messages they have sent before are received by $e^k + \delta$. Thus between time $\mathit{beg}^k + t_{\mathrm{del}} + \delta$ and time ready^{k+1}, no properly sent message from p can be received in q's buffer, with $q \in \mathcal{P}_j(\mathit{ready}^{k+1})$. By time $\mathit{beg}^k + t_{\mathrm{del}} + \delta + R(1+\varrho)$, q's buffer is empty (line 11). We have to show that (1) $\mathit{beg}^k + t_{\mathrm{del}} + \delta + R(1+\varrho) < \mathit{ready}^{k+1}$. The lower bound for ready^{k+1} is obtained as follows. Let p be the first process that properly terminates round k and let it be the process with the fastest clock. It will send $(\textsc{tick}, k+1)$ when its clock reads $(k+1)\,P$. Consequently, $\mathit{ready}^{k+1} \geq \mathit{beg}^k + \frac{P-A}{1+\varrho}$. From (1), we have to show that $t_{\mathrm{del}} + \delta + r < \frac{P-A}{1+\varrho}$. From constraints on P, and A it follows that we have to show that $t_{\mathrm{del}} + \delta + r < \frac{(t_{\mathrm{del}}+\delta)\cdot(1+\varrho)+R(1+\varrho)}{1+\varrho} = t_{\mathrm{del}} + \delta + R$ which is obvious from the definition of R.

(C) Straightforward from Lemma 5

(T) As no rules (except line `11`) are properly executed by processes in $\mathcal{C}_k$ between e^k and ready^{k+1}, no messages are sent by processes in $\mathcal{P}_j(\mathit{ready}^{k+1})$ within this interval. By Lemma 7, no process properly sends a $(\textsc{tick}, \ell)$ message, with $\ell \neq k+1$, within $[\mathit{ready}^k, \mathit{fast\text{-}go}^k + \delta]$. Finally by the induction assumptions, no process has properly sent a message $(\textsc{tick}, \ell)$ for $\ell \geq k$ before ready^k. □

Lemma 14. *For every round k it holds that $e^{k+1} - \mathit{ready}^k \leq j$.*

Theorem 3. *For every round k it holds that $\mathit{end}^k - \mathit{beg}^k \leq t_{\mathrm{del}}$.*

Proof. From Lemma 14 it follows that if $p \in \mathcal{P}_j(t)$ it holds that $p \in \mathcal{O}_k$ for the latest resynchronization period with $e^k \leq t$. Process p's round number is thus greater than k at time t and if it follows its algorithm until e^ℓ for some $\ell > k$ it sets its round number to $\ell + 1$ by then (Lemma 10) and thus, after e^k has a round number greater than k as long it remains obedient. Thus, it never again properly sends $(\textsc{tick}, k)$ after e^k, such that by Lemma 2 and Lemma 13 no process properly terminates round k after e^k; by the definition of end^k the theorem follows. □

We have seen that the collective update of the round numbers is ensured which is fundamental for round based clock synchronization algorithms. Based upon it one can show the following properties of the local bounded drift clocks.

Algorithm 2 Initialization Algorithm

```
variables
    Buffer^0[n] ← FALSE
    sent^0 ∈ {TRUE, FALSE} ← FALSE

on external start event do
    if sent^0 = FALSE then
        send (START) to all
    sent^0 ← TRUE

on receipt of message (START) sent by process q do
    Buffer^0[q] ← TRUE
    if |{r : Buffer^0[r]}| ≥ f + 1 then
        if sent^0 = FALSE then
            send (START) to all
        sent^0 ← TRUE
    if |{r : Buffer^0[r]}| ≥ n − f then
        C(t) ← A
        k ← 1                                   // start clock
```

Theorem 4 (Precision). *For all real-times t and for any two processes $p, q \in \mathcal{P}_j(t)$ it holds that*
$|C_p(t) - C_q(t)| \leq D_{\max}$, *with* $D_{\max} \triangleq \frac{P}{1+\varrho} \cdot dr + \frac{A}{(1+\varrho)^2} + \frac{t_{\text{del}}\,(1+\varrho\,(2+\varrho))}{1+\varrho}$

Theorem 5 (Accuracy). *For any process p and for any two real-times s and e with $p \in$ Obedient$(s, e) \wedge (e - s) > j$ it must hold for any two real-times $t_1, t_2 \in [s + j, e]$, $0 \leq t_1 < t_2$, that*

$$\frac{t_2 - t_1}{a} - b \leq C_p(t_2) - C_p(t_1) \leq (t_2 - t_1)\, c + d$$

with

$$a = 1 + \varrho \qquad b = 0$$
$$c = \frac{P\,(1+\varrho)}{P - A - t_{\text{del}}\,(1+\varrho)} \qquad d = P - \frac{P - A - t_{\text{del}}\,(1+\varrho)}{(1+\varrho)^2}$$

5 Initialization

For initial synchronization, it is usually assumed that all processes of the system are up and listening to the network when the algorithm is started [9, 10]. In systems where processes boot at unpredictable times, it was shown in [19] that this assumption can be dropped. In this paper, we consider the classic case and propose an initialization protocol which requires that there are sufficiently many processes following their protocol during the initialization phase.

Definition 4 (Failure Model for Initialization). *Let t be the maximum real-time at which a process $p \in$ Obedient$(0, t)$, properly starts initialization. Then it holds that $|$Obedient$(0, t_b)| \geq n - f$ with $t_b = t + 2\delta + 2 \cdot (P - A)(1 + \varrho)$, and $\forall t' > t_b$: $|\mathcal{P}_m(t')| > n - f$.*

Algorithm 2 presents a protocol which established initial synchronization, i.e., ensures Invariant 1 when used in conjunction with Algorithm 1 for $k = 2$: All processes

in $\mathcal{P}_j(t_b)$ properly terminate round 0 within t_{del} of each other. However, since these processes may start synchronization whenever they want, the initial synchronization period is is not bounded in size. This is only given for the first resynchronization such that starting with the second resynchronization (which can be shown to terminate before t_b) our properties (S), (N), (B), (C), and (T) hold. (More detailed analysis is given in [17].)

6 Related work

From the failure model perspective, the problem we solve is different from the Byzantine-tolerant self-stabilization version of clock synchronization [11, 12]. There, all the processes start with a possibly corrupted state, and eventually converge toward a safe state in which all processes have a synchronized clock. From the properties our algorithm achieves, we provide precise expressions on how many faults may occur in the system such that still perpetual clock synchronization is possible, which is in sharp contrast with self-stabilization.

The closest work to ours is the one of Barak et al. [14]. Their synchronization algorithm assumes that processes alternate between correct behaviors and faulty ones, and that no more than f processes can fail during sliding window of length θ. Differently from our solution, their resynchronization algorithm uses a convergence function similar to the differential fault-tolerant midpoint function of Fetzer and Cristian [20]. Maximal drift of logical clocks, ϱ, is very close to the one of hardware clocks, which shows the adequacy of that convergence function for maximizing logical clock accuracy. However, the weakness of their algorithm lies in the way clock synchronization is achieved. Whenever some process p decides to start a resynchronization phase, p asks all the processes to send their current clock values, which enables p to estimate the "system time". It is not hard to see that in case of Byzantine failures, resynchronizations can be invoked infinitely often with the main consequence of overloading processors and communication links which makes it difficult to guarantee some upper-bound on communication delays, as well as on the maximal error reading estimates, which has clearly a dramatic impact of convergence functions, and thus on the clock synchronization algorithm as the achievable precision depends on the timing uncertainty of the system [21]. Modifying their algorithm to prevent such behavior does not seem trivial. An idea would be to reject/ignore too early clock synchronization messages, but this would postpone recovery, and probably would have severe impact on the correctness proof. In contrast to their solution, ours does not compute a new clock based on the clock values of other processes but based on the receipt of a minimum number of synchronization messages; some of which must have been sent by "correct" processes, preventing thus abusive release clock resynchronizations. Finally, regarding fault turn-over rate, we improve the results by Barak et al. [14] by a factor of approximately 3.

Anceaume et al. [13] present an ad-hoc solution to the clock synchronization problem for the particular case where $f = 1$ and $n \geq 4$. The present work is a generalization of that result by considering an unvalued variable f.

Open Problems. We proposed (simple) mechanisms to transform a clock synchronization algorithm tolerant to permanent Byzantine failures into an algorithm in which all

processes may recover after Byzantine failures. This transformation takes advantage of the inherent properties of the problem we address, namely, data have a limited duration of validity, or can be refreshed periodically. We conjecture that it is possible to design automatic transformations for all distributed algorithms that manipulate evanescent variables.

References

1. 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
2. Simons, B., Welch, J., Lynch, N.: An overview of clock synchronization. In: Fault Tolerant Distributed Computing. Volume 448 of LNCS. (1990) 84–96
3. Schneider, F.B.: Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Cornell University, Dept. of Computer Science (1987)
4. Anceaume, E., Puaut, I.: Performance evaluation of clock synchronisation algorithms. Technical Report 3526, INRIA (1998)
5. Schmid, U., ed.: Special Issue on The Challenge of Global Time in Large-Scale Distributed Real-Time Systems. J. Real-Time Systems 12(1–3) (1997)
6. Guzella, R., Zatti, S.: The accuracy of clock synchronization achieved by Tempo in Berkeley Unix 4.3BSD. IEEE Transactions on Software Engineering **15**(7) (1989) 847–853
7. Cristian, F., Aghili, H., Strong, R.: Clock synchronization in the presence of omission and performance failures, and joins. In: Proc. of the 15th Int'l Symposium on Fault Tolerant Computing, IEEE (1986)
8. Halpern, J., Simons, B., Strong, R., Dolev, D.: Fault-tolerant clcok synchronization. In: Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, ACM (1984) 89–102
9. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM **34**(3) (1987) 626–645
10. Lundelius Welch, J., Lynch, N.: A new fault tolerant algorithm for clock synchronization. Information and Computation **77**(1) (1988) 1–36
11. Daliot, A., Dolev, D., Parnas, H.: Linear time byzantine self-stabilizing clock synchronization. In: Proceedings of the 7th International Conference on Principles of Distributed Systems. Volume 3144 of LNCS., Springer Verlag (2003) 7–19
12. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM **51**(5) (2004) 780–799
13. Anceaume, E., Delporte-Gallet, C., Fauconnier, H., Hurfin, M., Le Lann, G.: Designing modular services in the scattered byzantine failure model. In: 3rd International Symposium on Parallel and Distributed Computing. (2004) 262–269
14. 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
15. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3) (1982) 382–401
16. Lamport, L., Melliar-Smith, P.M.: Synchronizing clocks in the presence of faults. Journal of the ACM **32**(1) (1985) 52–78
17. Anceaume, E., Delporte-Gallet, C., Fauconnier, H., Hurfin, M., Widder, J.: Clock synchronization in the Byzantine-recovery failure model. Technical Report 59/2007, Technische Universität Wien, Institut für Technische Informatik (2007)

18. Srikanth, T., Toueg, S.: Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distributed Computing **2** (1987) 80–94
19. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distributed Computing **20**(2) (2007) 115–140
20. Fetzer, C., Cristian, F.: An optimal internal clock synchronization algorithm. In: Proceedings of the 10th Annual Conference on Computer Assurance. (1995) 187–196
21. Lundelius, J., Lynch, N.: An upper and lower bound for clock synchronization. Information and Control **62** (1984) 190–240