# Linear Time Byzantine Self-Stabilizing Clock Synchronization

Ariel Daliot[1], Danny Dolev[1⋆], and Hanna Parnas[2]

[1] School of Engineering and Computer Science, The Hebrew University of Jerusalem, Israel. {adaliot,dolev}@cs.huji.ac.il
[2] Department of Neurobiology and the Otto Loewi Minerva Center for Cellular and Molecular Neurobiology, Institute of Life Science, The Hebrew University of Jerusalem, Israel. hanna@vms.huji.ac.il

**Abstract.** Awareness of the need for robustness in distributed systems increases as distributed systems become an integral part of day-to-day systems. Tolerating Byzantine faults and possessing self-stabilizing features are sensible and important requirements of distributed systems in general, and of a fundamental task such as clock synchronization in particular. There are efficient solutions for Byzantine non-stabilizing clock synchronization as well as for non-Byzantine self-stabilizing clock synchronization. In contrast, current Byzantine self-stabilizing clock synchronization algorithms have exponential convergence time and are thus impractical. We present a linear time Byzantine self-stabilizing clock synchronization algorithm, which thus makes this task feasible. Our deterministic clock synchronization algorithm is based on the observation that all clock synchronization algorithms require events for re-synchronizing the clock values. These events usually need to happen synchronously at the different nodes. In these solutions this is fulfilled or aided by having the clocks initially close to each other and thus the actual clock values can be used for synchronizing the events. This implies that the clock values cannot differ arbitrarily, which necessarily renders these solutions to be non-stabilizing. Our scheme suggests using a tight pulse synchronization that is uncorrelated to the actual clock values. The synchronized pulses are used as the events for re-synchronizing the clock values.

## 1 Introduction

Overcoming failures that are not predictable in advance is most suitably addressed by tolerating Byzantine faults. It is the preferred fault model in order to seal off unexpected behavior within limitations on the number of concurrent faults. Most distributed tasks require the number of Byzantine faults, $f$, to abide by the ratio of $3f < n$, where $n$ is the network size. See [**?**] for impossibility results on several consensus related problems such as clock synchronization. Additionally, it makes sense to require such systems to resume operation after serious unpredictable events without the need for an outside intervention or restart of the system from scratch. E.g. systems may occasionally experience

⋆ This research was supported in part by Intel COMM Grant - Internet Network/Transport Layer & QoS Environment (IXA)

short periods in which more than a third of the nodes are faulty or messages sent by all nodes may be lost for some time. Such transient violations of the basic fault assumptions may leave the system in an arbitrary state from which the protocol is required to resume in realizing its task. Typically, Byzantine algorithms do not ensure convergence in such cases. Byzantine algorithms focus on merely preventing Byzantine faults from notably shifting the system state away from the goal. They sometimes make strong assumptions on the initial state. A self-stabilizing algorithm overcomes this limitation by converging within finite time to a correct state from any initial state. Thus, even if the system loses its consistency due to a transient violation of the basic fault assumptions (e.g. more than a third of the nodes being faulty, network disconnected, etc.) then once the system is back within the assumption boundaries the protocol will successfully realize the task, irrespective of the resumed state of the system. For a short survey of self-stabilization see [**?**] and for an extensive study see [**?**].

The current paper addresses the problem of synchronizing clocks in a distributed system. There are several efficient algorithms for self-stabilizing clock synchronization withstanding crash faults (see [**?**,**?**,**?**] or other variants of the problem [**?**,**?**]). There are many efficient classic Byzantine clock synchronization algorithms (for a performance evaluation of clock synchronization algorithms see [**?**]), however strong assumptions on the initial state of the nodes are typically made, such as assuming all clocks are initially synchronized ([**?**,**?**,**?**]) and thus are not self-stabilizing. On the other hand, self-stabilizing clock synchronization algorithms can initiate with arbitrary values which can have a cost in the convergence times or in the severity of the faults contained. There are surprisingly few self-stabilizing solutions facing Byzantine faults ([**?**]), which additionally have unpractical convergence times. Note that self-stabilizing clock synchronization has an inherent difficulty in estimating real-time without an external time reference due to the fact that non-faulty nodes may initialize with arbitrary clock values. Thus self-stabilizing clock synchronization aims at reaching a stable state from which clocks proceed synchronously at the rate of real-time and not necessarily estimate real-time (assuming that nodes have access to physical timers that proceed close to real-time rate). Many applications utilizing the synchronization of clocks do not really require the exact real-time notion (see [**?**]). In such applications, agreeing on a common clock reading is sufficient as long as the clocks progress within a linear envelope of any real-time interval.

We present a protocol with the following property: should the system be initialized with clocks that hold values that are close to real-time then the clocks stay synchronized while attaining similar real-time accuracy, precision and time complexity as non-stabilizing clock synchronization protocols. Should the system initialize with arbitrary clock values or recover from any transient faults then the clocks synchronize very fast and proceed at real-time rate with high precision.

The protocol we present significantly improves upon existing Byzantine self-stabilizing clock synchronization algorithms by reducing the time complexity from expected exponential ([**?**]) to deterministic $O(f)$. The comparably low complexity is achieved by focusing on a deterministic Byzantine self-stabilizing algorithm for pulse synchronization. The synchronized pulses progress at a pace that allows the execution of a Byzantine Strong Consensus protocol on the clock values in between pulses, thus obtaining a common clock reading.

A special challenge in self-stabilizing clock synchronization is the clock wrap around. In non-stabilizing algorithms having a large enough integer eliminates the problem for any practical concern. In self-stabilizing schemes a transient failure can cause nodes to initialize with arbitrarily large clocks, surfacing the issue of the clock bounds. The clock synchronization schemes described above handles this wrap around.

Having access to an outside source of real-time is useful, though it introduces a single point of failure. Our approach is useful in such a case to overcome periods in which the outside source fails in order to maintain a consistent system state.

## 2 Model and Problem Definition

The environment is a network of processors (nodes) that communicate by exchanging messages. Individual nodes have no access to a central clock and there is no global pulse system. The hardware clocks (referred to as the *physical timers*) of correct nodes have a bounded drift rate, $\rho$, from real-time. The communication network does not guarantee any order on messages.

The network and/or all the nodes can behave arbitrarily, though eventually the network performs within the defined assumption boundaries in which at most $f$ out of the $n$ nodes may behave arbitrarily.

**Definition 1.** *The* **network assumption boundaries** *are:*

1. *Message passing allowing for an authenticated identity of the senders.*
2. *At most $f$ of the nodes are faulty.*
3. *Any message sent by any non-faulty node will eventually reach every non-faulty node within $\delta$ time units.*

**Definition 2.** *A node is* **correct** *at times that it complies with the following conditions:*

1. *Obeys a global constant $0 < \rho_{glob} << 1$, such that for every Newtonian time interval $[u, v]$, $(1-\rho_{glob})(v-u) \leq$ 'physical timer'$(v)-$ 'physical timer'$(u) \leq (1+\rho_{glob})(v-u)$. Hereafter $\rho_{glob}$ is denoted by $\rho$ (typically $\rho \approx 10^{-6}$).*
2. *Operates according to the instructed protocol.*

A node is considered **faulty** if it violates one or more of the above. A faulty node recovers from its faulty behavior if it resumes obeying the conditions of a correct node. For consistency reasons the recovery is not immediate, but rather takes a certain amount of time during which the node is still considered faulty although it behaves correctly[3].

**Basic notations:**
We use the following notations to define the quality of the solution, though nodes do not need to maintain all of them as variables.

- $Clock_i$ (the clock of node $i$) is a function of node $p_i$'s physical timer that returns a value in the range 0 to $M-1$. Thus $M-1$ is the maximal value a clock can hold. The clock is incremented every time unit. $Clock_i(t)$ denotes the value of the clock of node $p_i$ at real-time $t$.

[3] For example, a node may recover during the Byzantine Consensus procedure and violate the validity condition if considered correct immediately.

- A "*pulse*" is an internal event for the re-synchronization of the clocks, ideally every *cycle* time units. A cycle is the time interval between two successive pulses that a node invokes.
- $\sigma$ represents the upper bound on the real-time between the invocations of the pulses of different correct nodes (*tightness of pulse synchronization*).
- $\gamma$ is the target upper bound on the difference of clock readings of any two correct clocks at any real-time. Our protocol achieves $\gamma = 5d + c \cdot \rho$, for some constant $c$.
- Let $a, b, g, h \in R^+$ be constants that define the linear envelope bound of the clock progression rate on any real-time interval.
- $\Psi_i(t_1, t_2)$ is the amount of clock time elapsed on the clock of node $p_i$ during a real-time interval $[t_1, t_2]$ within which $p_i$ was continuously correct. The value of $\Psi$ is not affected by any wrap around of $clock_i$ during that period.
- $d \equiv \delta + \pi$, where $\pi$ is the upper bound on message processing time.

Thus, $d$ is an upper bound on the elapsed real-time from sending a message by a non-faulty node until it is received and processed by every non-faulty node.

Using the above notations, a recovered node can be considered correct once it goes through a complete synchronization process, which is guaranteed to happen within $cycle + \text{BYZ\_}time$ of correct behavior, where BYZ_*time* is the time to complete the Byzantine Consensus algorithm.

**Definition 3. Clock_state**

- *The* **clock_state** *of the system at real-time $t$ is given by:* $clock\_state(t) \equiv (clock_0(t), \ldots, clock_{n-1}(t))$.
- *The systems is in a* **synchronized clock_state** *at real-time $t$ if* $\forall correct\ p_i, p_j,\ |clock_i(t) - clock_j(t)| \leq \gamma$ *or*[4] $|clock_i(t) - clock_j(t)| \geq M - \gamma$.

**Definition 4. The Self-Stabilizing Clock Synchronization Problem**
*As long as the system is within the assumption boundaries:*

**Convergence:** *Starting from an arbitrary state, $s$, the system reaches a synchronized clock_state after a finite time.*

**Closure:** *If $s$ is a synchronized clock_state of the system at real-time $t_0$ then* $\forall real\ time\ t \geq t_0$,

1. *clock_state(t) is a synchronized clock_state,*
2. *"Linear Envelope": for every correct node, $p_i$,*

$$a \cdot [t - t_0] + b \leq \Psi_i(t_0, t) \leq g \cdot [t - t_0] + h.$$

## 3 Self-Stabilizing Byzantine Clock Synchronization

A major challenge of self-stabilizing clock synchronization is to ensure clock synchronization even when nodes may initialize with arbitrary clock values. This, as mentioned before, requires handling the wrap around of clock values. The

[4] The second condition is a result of dealing with bounded clock variables.

algorithm we present employs as a building block an underlying Byzantine self-stabilizing pulse synchronization procedure. In the pulse synchronization problem nodes invoke pulses regularly, ideally every *cycle* time units, and the goal is to do so in tight synchrony. To synchronize their clocks, nodes initiate at every pulse a Strong Byzantine Consensus to agree on the clock value to be associated with the time of the next pulse event[5]. Once pulses are synchronized, then the consensus results in synchronized clocks. The basic algorithm uses strong consensus to ensure that once correct clocks are synchronized at a certain pulse and thus enter the consensus procedure with identical values, then they terminate with the same identical values and thus keep the progression of clocks continuous and synchronized[6].

### 3.1 The Basic Algorithm

The basic algorithm is essentially a self-stabilizing version of the Byzantine clock synchronization algorithm in [**?**]. We call it Pulse-Clock-Synch. The agreed clock time of the next synchronization is denoted by ET (short for Expected Time, as in [**?**]). Synchronization of clocks is targeted to happen every *cycle* time units, unless the pulse is invoked earlier[7].

**Pulse-Clock-Synch**
**at** "$pulse$" event
  **begin**
    $Clock = ET$;
    Abort possible running instance of Pulse-Clock-Synch and reset all buffers;
    Wait $\sigma(1 + 2\rho)$ time units;
    $Next\_ET$ = Strong-Byz-Consensus($(ET + cycle)$ mod $M$);
    $Clock = (Clock + Next\_ET - (ET + cycle))$ mod $M$;
    $ET = Next\_ET$;
  **end**

The internal pulse event is delivered by the pulse synchronization procedure. This event stops all on-going agreements and previous invocations of Pulse-Clock-Synch and resets all buffers. The pulse synchronization procedure ensures that once pulses are synchronized, the Pulse-Clock-Synch are invoked within $\sigma$ real-time units[8] of the pulse invocations at all correct nodes.

The "Wait" intends to make sure that all correct nodes enter the Byzantine Consensus after the invocation of the pulse event at all correct nodes, without remnants of past invocations. Existence of past remnants may happen only when the system is not yet synchronized or it is not within the assumption boundaries.

A correct node joins a Byzantine Consensus only concomitant to an internal pulse event, as instructed by the Pulse-Clock-Synch. The Strong-Byzantine-

[5] It is assumed that the time between successive pulses is sufficient for a Byzantine Consensus algorithm to initiate and terminate in between.
[6] The Pulse Synchronization building block does not use the value of the clock to determine its progress, but rather intervals measured on the physical timer.
[7] *cycle* has the same function as PER in [**?**].
[8] The pulse synchronization presented achieves $\sigma = 2d$.

Consensus is intended to reach consensus on the next value of $ET$. One can use a synchronous agreement algorithm with rounds of size $(\sigma + d)(1 + 2\rho)$ or asynchronous style agreement in which a node waits to get $n - f$ messages of the previous round before moving to the next round. We assume the use of a Byzantine Consensus algorithm tolerating $f$ faults when $n \geq 3f + 1$.

The Strong-Byzantine-Consensus also ensures that when the clocks are synchronized the wrap around of clocks happens at all correct nodes within a short time and at the same cycle.

The posterior clock adjustment (following the consensus) adds to the clock value the elapsed time since the pulse and until the end of the consensus, as if the time at the pulse was in accordance with the relevant time reflected by the consensus for the next pulse. This intends to expedite the time for reaching synchronized clocks in the case in which the node's initial value of $ET$ did not agree with the value at the rest of the correct nodes.

Notice that when the system is back within the assumption boundaries, following a chaotic state, pulses may arrive to different nodes at arbitrary times, and nodes' $ET$ and clocks may differ arbitrarily. At that time not all nodes will join the Byzantine Consensus and no consistent resultant value can be guaranteed. Once the pulses become synchronized (guaranteed by the pulse synchronization procedure to happen within a single cycle) all correct nodes will join the same execution of the Byzantine Consensus and will agree on the clock value of the next synchronization. From that time on, as long as the system stays within the assumption boundaries the clocks remain synchronized, similar to [**?**].

Note that instead of simply setting the clock value to $ET$ we could use some Clock-Adjust procedure (cf. [**?**]), which receives a parameter indicating the target value of the clock. The procedure runs in the background, it speeds up or slows down the clock rate to reach the adjusted value within a specified period of time. The procedure handles clock wrap around.

**Theorem 1.** *Pulse-Clock-Synch solves the Self-Stabilizing Clock Synchronization Problem in the presence of at most $f$ Byzantine nodes, where $n \geq 3f + 1$.*

*Proof.* Assume that the Pulse Synchronization procedure that invokes the pulses solves the Self-Stabilizing Pulse Synchronization Problem, as defined in Section **??**, in the presence of at most $f$ Byzantine nodes, where $n \geq 3f + 1$. Assume that $cycle_{\min} \geq 2\sigma + byz\_time$, where $byz\_time$ is the maximal time it takes to complete the consensus algorithm.

**Convergence:** Let the system be within the assumption boundaries, in an arbitrary state, $s$, with the nodes holding arbitrary clock values. The pulse synchronization procedure is self-stabilizing, thus, independent of the system's initial state, within a finite time the pulses are invoked regularly and synchronously with a tightness of $\sigma$. At every pulse all remnants of previously invoked consensus algorithms are flushed by all the correct nodes. A correct node does not initiate or join the consensus algorithm before waiting $\sigma(1+2\rho)$ time units, hence not before all correct nodes have invoked a pulse and subsequently flushed their buffers. All correct nodes will join the consensus, thus the consensus algorithm will initiate and terminate successfully.

At termination of the first instance of the consensus algorithm following the synchronization of the pulses, all correct nodes agree on the clock value to be

held at the next pulse invocation. Subsequently, all correct nodes adjust their clocks, post factum, according to the agreed $ET$. Note that this posterior adjustment of the clocks does not affect the time span until the invocation of the next pulse but rather updates the clocks concomitantly to and in accordance with the newly agreed $ET$. This has an effect if the correct nodes entered the consensus with differing values. Hence if all correct nodes enter the consensus with the same $ET$ then the adjustment equals zero. Since all correct pulses arrived within $\sigma$ of each other, after the posterior clock adjustment of the last correct node, all correct clocks are within $\gamma_1 = \sigma(1+2\rho) + \mathrm{b}yz\_end \cdot 2\rho$,[9] where $\mathrm{b}yz\_end$ is the maximal span of time from the first pulse until the last correct node completes the posterior clock adjustment. Correct clocks will continue to drift apart at a rate of $2\rho$ until the next clock adjustment, which will take place in about a *cycle* time. The precision of the clocks (the bound on the clock differences), $\gamma \geq \gamma_1$, equals the bound on the precision of the pulse arrival and the accumulated skew on the upper bound cycle length. This concludes the Convergence condition.

**Closure:** Let the system be in a synchronized clock_state. Consider first the case in which all correct nodes hold the same value for $ET$. In this case, each correct node resets its clock when the pulse arrives and doesn't adjust its clock after the consensus, since the consensus value will be the same as the value it entered the consensus with. To simplify the discussion assume that no wrap around of any correct clock takes place during the time that the pulse arrives at the first correct node and until it is invoked at the last correct node. Right after the pulse is invoked at the last correct node and it's subsequent clock adjustment, all correct clocks are within $\gamma_0 = \sigma(1+2\rho)$ of each other.

From that point on clocks of correct nodes drift apart at a rate of $2\rho$ of each other. As long as no wrap around of the clocks takes place and no pulse arrives at any correct node, the clocks are at most $\gamma_0 + \Delta T \cdot 2\rho$ apart, where $\Delta T$ is the real-time elapsed since the invocation of the pulse at the first correct node. The forth-coming pulses will be invoked at all correct nodes within at most $cycle + \sigma$ time, thus the bound on the clock difference of correct nodes, before the first pulse is invoked again is $\gamma_0 + cycle \cdot 2\rho$.

When a correct node resets its clock once a pulse arrives, the maximal clock adjustment is the maximal difference between its current clock value and the value of $ET$. We discuss this difference, denoted by $ADJ$, later on in the proof and in Section **??**. The clocks will continue to drift apart until the last correct node receives its "pulse" event. Therefore, the maximal clock difference, as long as no clock wrap around takes place is bounded by $ADJ + \gamma_0 + cycle \cdot 2\rho + \sigma(1+2\rho)2\rho \leq \gamma$.

We now consider the case that a clock wrap around takes place at some $\Delta T$ time after the pulse is invoked at the last correct node. From the discussion above we learn that at the moment prior to the first correct clock wrap around, the correct clocks are at most $\gamma_0 + \Delta T \cdot 2\rho$ apart. Therefore, all correct clocks will wrap around within $\gamma_0 + \Delta T \cdot 2\rho$. During this time any two correct clocks, $i, j$, satisfy $|clock_i(t) - clock_j(t)| \geq M - (\gamma_0 + \Delta T \cdot 2\rho)(1+2\rho) \geq M - \gamma$. Note,

[9] The $2\rho$ is the maximal drift rate between any two correct clocks, whereas $\rho$ is their drift with respect to real-time.

that a similar discussion can show that even if clocks wrap around in-between the pulse invocations at the correct nodes the systems stays in a synchronized clock_state.

The case that remains to consider is when at a synchronized clock_state not all correct nodes hold the same value of $ET$. At the end of the convergence, as we proved above, all correct clocks are $\gamma_1$ apart. Following the first consensus all the discussion above remains, and from that point on they will remain in a synchronized clock_state, and the only time their $ET$s differ is when some received the "pulse" event and some are about to receive it. This completes the proof of the first Closure condition.

Note that $\Psi_i$, defined in Section **??**, represents the actual deviation of an individual correct clock ($p_i$,) over a real-time interval. The accuracy of the clocks is the bound on the deviation of correct clocks over a real-time interval. The clocks are repeatedly adjusted at every pulse. It is required that the pulses progress within a linear envelope of any real time interval (see Section **??**). Thus we need to show that the adjustment to the clocks at every pulse is a linear function of the length of the cycle. The accuracy equals the bound on $|t_{pulse} - ET_{pulse}|$, where $t_{pulse}$ is the clock value at the pulse at the moment prior to the adjustment of the clock to $ET_{pulse}$. The upper and lower bounds on the value $t_{pulse}$ is determined by the bounds $cycle_{\min}$ and $cycle_{max}$ on the cycle length. Thus,

$$ET_{prev-pulse} + cycle_{\min} \cdot (1 - \rho) \leq t_{pulse} \leq ET_{prev-pulse} + cycle_{\max} \cdot (1 + \rho).$$

The adjustment to a correct clock, $ADJ$, is thus bounded by:
$|ET_{prev-pulse} + cycle_{\min} \cdot (1 - \rho) - ET_{pulse}| \leq ADJ \leq$

$$|ET_{prev-pulse} + cycle_{\max} \cdot (1 + \rho) - ET_{pulse}|,$$

which implies,

$$|ET_{prev-pulse} + cycle_{\min} \cdot (1 - \rho) - (ET_{prev-pulse} + cycle)| \leq ADJ \leq$$
$$|ET_{prev-pulse} + cycle_{\max} \cdot (1 + \rho) - ET_{prev-pulse} + cycle)|,$$

which implies,

$$|cycle_{\min} \cdot (1 - \rho) - cycle| \leq ADJ \leq |cycle_{max} \cdot (1 + \rho) - cycle|.$$

As can be seen, the bound on the adjustment to the clock is linear in the effective cycle length. The bounds on the effective cycle length are guaranteed, by the pulse synchronization procedure, to be linear in the default cycle length. Thus the accuracy of the clocks are within a linear envelope of any real-time interval. The actual values of $cycle_{\min}$ and $cycle_{max}$ are specifically determined by the specific pulse synchronization procedure used. This concludes the Closure condition.

Thus the algorithm is self-stabilizing and performs correctly with $f$ Byzantine nodes, when $n \geq 3f + 1$. □

### 3.2 An Additional Self-stabilizing Clock Synchronization Algorithm

We end the section by suggesting a simple additional Byzantine self-stabilizing clock synchronization algorithm using pulse synchronization as a building block.

Our second algorithm resets the clock at every pulse[10]. This approach has the advantage of not needing any agreement algorithm. This version is useful, for example, when *cycle* is on the order of $M$.

**NOADJUST-CS**
**at** "$pulse$" event
  **begin**
    Clock = 0;
  **end**

The algorithm has the disadvantage that the real-time span for the clock to reach $M$ is bounded by the cycle length. This can be counteracted by using a very large cycle but this enhances the effect of the clock skew, which negatively affects the precision and the accuracy.

## 4 Self-Stabilizing Byzantine Pulse Synchronization

The nodes execute this procedure in the background. The procedure ensures that different nodes invoke pulses in a close time proximity ($\sigma$) of each other. Pulses should be invoked regularly. The pulse synchronization should invoke the pulses within linear envelope of real-time intervals.

**Basic notations:**
In addition to the definitions of Section **??** we use the following notations to define the quality of the solution, though nodes do not need to maintain them as variables.

- $\psi_i(t_1, t_2)$ is the number of pulses a correct node $p_i$ invoked during a real-time interval $[t_1, t_2]$ within which $p_i$ was continuously correct.
- Let $a', b', g', h' \in R^+$ be constants that define the linear envelope bound on the ratio between all real-time intervals and every $\psi_i$ in those intervals.
- $\phi_i(t) \in R^+ \cup \{\infty\}$, $0 \leq i \leq n$, denotes the elapsed real-time since the last time node $p_i$ invoked a pulse. For a node, $p_j$, that has not invoked a pulse since the initialization of the system, $\phi_j(t) \equiv \infty$.

**Basic definitions:**

- The **pulse_state** of the system at real-time $t$ is given by: $pulse\_state \equiv (\phi_0(t), \ldots, \phi_{n-1}(t))$.
- Let $G$ be the set of all possible pulse_states of a system $S$.
- A set of nodes, $N$, are called **pulse-synchronized** at real-time $t$ if

$$\forall p_i, p_j \in N, |\phi_i(t) - \phi_j(t)| \leq \sigma.$$

[10] This approach was suggested also by Shlomi Dolev.

– $s \in G$ is a **synchronized pulse_state** of the system at real-time $t$ if the set of correct nodes are pulse-synchronized at some real-time $t_{syn}$ in the interval $[t, t + \sigma]$.

**Definition 5. The Self-Stabilizing Pulse Synchronization Problem**

*As long as the system is within the assumption boundaries:*
**Convergence:** *Starting from an arbitrary state, s, the system reaches a synchronized pulse_state after a finite time.*
**Closure:** *If s is a synchronized pulse_state of the system at real-time $t_0$ then $\forall$real time $t \geq t_0$,*

1. *pulse_state(t) is a synchronized pulse_state,*
2. *"Linear Envelope": for every correct node, $p_i$,*

$$a' \cdot [t - t_0] + b' \leq \psi_i(t, t_0) \leq g' \cdot [t - t_0] + h'.$$

3. *$\exists\, cycle_{\min}, cycle_{\max}$ such that $1 \leq \psi_i(t, t_0)$, for every $t - t_0 \geq cycle_{\max}$, and $1 \geq \psi_i(t, t_0)$, for every $t - t_0 \leq cycle_{\min}$.*

The third condition intends to bound the rate of pulses between a minimal and a maximal rate.

### 4.1 The mode of operation of the Pulse Synchronization procedure

The Byzantine self-stabilizing pulse synchronization procedure presented is called SS-Pulse-Synch. A cycle is the time interval between two successive pulses a node invokes. The default value of *cycle* is the ideal length of the cycle. The actual real-time length of a cycle may slightly deviate from the value *cycle* in consequence of the clock drifts, uncertain message delays and behavior of faulty nodes. In Theorem **??** the extent of this deviation is explicitly shown, defining the bounds of the linear envelope. Toward the end of its cycle, every correct node targets at synchronizing its forthcoming pulse invocation with the pulse of the other nodes. It does so by sending a **Propose-Pulse** message to all nodes. These messages (or a reference to the sending node) are accumulated at each correct node until it invokes a pulse and deletes these messages (or references). We say that two Propose-Pulse messages are **distinct** if they were sent by different nodes.

When a node accumulates at least $f + 1$ distinct Propose-Pulse messages it also triggers a Propose-Pulse message. Once a node accumulates $n - f$ distinct Propose-Pulse messages it invokes the pulse. The input to the procedure is $cycle, n$ and $f$.

**SS-Pulse-Synch**
**if** ($cycle\_countdown\_is\_0$) then */* endogenous message */*
  send "Propose-Pulse" message to all;
  $cycle\_countdown\_is\_0$='False';
**if** received "Propose-Pulse" messages from $f + 1$ distinct nodes then
*/* triggered message */*
  send "Propose-Pulse" message to all;
**if** received "Propose-Pulse" messages from $n - f$ distinct nodes then

```
                                                    /* invoke pulse */
begin
   invoke "pulse" event;
   cycle_countdown = cycle;
   flush "Propose-Pulse" message counter;
   ignore "Propose-Pulse" messages for 2d(1 + 2ρ) time units;
end
```

We assume that a background process continuously reduces *cycle_countdown*. Once it reaches 0 (intended to make the node count approximately *cycle* time units on its physical timer), the background process resets the value back to *cycle* and invokes the SS-Pulse-Synch by setting *cycle_countdown_is_*0 to 'True'. A reset is also done if *cycle_countdown* holds a value that is out of range (not between 0 and *cycle*). The value is set back again to *cycle* in the algorithm once the "pulse" is invoked in order to prevent the system from blocking because of initializing or recovering with a wrong value in the *cycle_countdown* or in the program counters of the nodes. Note that on a premature execution of SS-Pulse-Synch the node does not flush its message counter.

Note that a node typically sends the message more than once within a cycle. This is done to prevent cases in which the node may be invoked in a state that leads to a deadlock.

When the system is not in a synchronized state a correct node may need to wait almost a cycle before others will join its proposal, but once all correct nodes invoke the pulses a correct node will need to wait a much shorter time, as we prove below.

**Theorem 2.** *SS-Pulse-Synch solves the Self-Stabilizing Pulse Synchronization Problem in the presence of at most $f$ Byzantine nodes, where $n \geq 3f + 1$.*

*Proof.* Assume that the system is within the assumption boundaries.
**Convergence:** Every correct node sends at least one Propose-Pulse messages in every one of its cycles because every correct node's cycle_countdown timer eventually reaches 0. Thus, in every real-time interval equal to the maximal length a cycle can extend, at least $n - f$ distinct Propose-Pulse messages are sent. Let a correct node accumulate $n - f$ distinct Propose-Pulse messages and thus invoke a pulse; then at least $n-f-f = n-2f$ of them must be from correct nodes. We assume that $n > 3f$, implying that at least $3f + 1 - 2f = f + 1$ of these messages must be from correct nodes. All correct nodes will receive these (at least) $f + 1$ distinct Propose-Pulse messages within $d$ time units of the time that the first correct node received the $n - f$ messages. Consequently, following the algorithm, they will also send Propose-Pulse messages. Within additional $d$ time units all correct nodes will receive at least $n - f$ distinct Propose-Pulse messages and thus invoke a pulse, at most $2d$ time units after the first node invoked its pulse. This concludes the Convergence requirement for $\sigma = 2d$.

Irrespective of the initial values of the various variables of the nodes, once the system is in the assumption boundaries, within a single cycle each node will reach a point at which it is not blocking and all its variables hold legal values, following which it invokes a pulse within a cycle and within $\sigma$ of the other correct nodes.

**Closure:** Let the system be in a synchronized pulse_state at the time immediately following the last correct node to invoke its pulse. Thus, all correct nodes have invoked their pulse, flushed the counters and reset their *cycle_countdown* timer within $2d$ time units of each other. They are also ignoring all Propose-Pulse messages $\sigma(1 + 2\rho)$ time units subsequent to their pulse and thus the last node's messages related to its pulse, are necessarily ignored by all correct nodes. Thus, no correct node will accumulate $f + 1$ distinct Propose-Pulse messages before at least one correct node sends an endogenous message the next time, irrespective of the behavior of the faulty nodes. No correct node will send that message before counting *cycle* time units following its pulse invocation. The node that sent its message will not invoke its pulse before it has accumulated $n - f$ distinct Propose-Pulse messages. Following the same arguments as for the Convergence, all correct nodes henceforth invoke their pulse within $2d$ real-time units of each other. Thus the system stays in a synchronized pulse_state and so the first Closure condition is satisfied.

To identify the shortest elapsed time a correct node may invoke a new pulse following its previous pulse invocation let us observe the following scenario: Let a node, $p$, invoke its pulse $2d$ real-time units after all other correct nodes. Next assume that the rest of the correct (fast) nodes reach their endogenous message point and send their messages. Assume that node $p$ receives $n - f$ such messages almost instantaneously and thus invokes its pulse at that point. The correct (fast) nodes then invoke their new pulses exactly $\frac{cycle}{1+\rho}$ real-time units after their previous pulses. Thus, node $p$ may invoke its new pulse $\frac{cycle}{1+\rho} - 2d$ real-time units subsequent to its former pulse. This determines the value of $cycle_{min}$. Being linear in the default *cycle* length this, thus, defines a lower linear envelope.

Equivalently, let a correct node invoke its pulse $2d$ real-time units before all other correct nodes. Let all the other correct (slow) nodes reach their endogenous message sending - this will happen at most $\frac{cycle}{1-\rho}$ real-time units after their previous pulses. Let it take another $d$ real-time units for these messages to reach our node, by which it will invoke its new pulse immediately. This yields an upper bound on a correct nodes cycle length of $\frac{cycle}{1-\rho} + 3d$ real-time units subsequent to its former pulse. This determines the value of $cycle_{max}$. Being linear in the default *cycle* length this, thus, defines an upper linear envelope. And so the second Closure condition is satisfied.

Thus the algorithm is self-stabilizing and performs correctly with $f$ Byzantine nodes, when $n \geq 3f + 1$. □

## 5 The Self-Stabilizing Byzantine Consensus Algorithm

The Strong Byzantine Consensus module can use many of the classical Byzantine Consensus algorithms. The self-stabilization does not introduce a major obstacle, because the algorithms terminate in a small number of rounds, and cycle can be set so it is before the next invocation of the algorithm. The only delicate point is to make sure that the algorithm doesn't cause the nodes to block or deadlock. Below we specify how to update the use the early stopping Byzantine Algorithm of Toueg, Perry and Srikanth [**?**] to address our needs. The nodes invoke the algorithm with their value of the next ET.

The original algorithm is synchronous. In our environment the nodes will clock the phases and instead of considering messages that should arrive within a given phase, they should consider messages that arrive by the end of the indicated phase on their own clock. When nodes invoke the procedure they consider also all messages in their buffers that were accepted prior to the invocation.

We use the following notations in the description of the consensus algorithm:
- A *phase* is a duration of $(\sigma + d)(1 + 2\rho)$ clock units on a node's clock.
- A *round* is a duration of two phases.
- A *broadcast* primitive is the primitive defined in [**?**] (see Appendix). Nodes issue an *accept* within the broadcast primitive.

The main differences of the protocol below from the original protocol of [**?**] are:
- Instead of the General use an imaginary node whose value is the clock values of the individual nodes.
- Agree on whether $n - f$ of the values are identical.
- The fact that the general is not counted as a faulty node requires running the protocol an extra round.

**Strong-Byz-Consensus**$(ET)$ invoked at node $p$:
  **initialize** the **broadcast** primitive;
  $broadcasters := \emptyset$; $v = 0$;
**phase** $= 1$ :
  **send**$(ET, 0)$ to all participating nodes;
**phase** $= 2$ :
  **if received** $(ET', 0)$ messages for from $n - f$ distinct nodes by
      the end of phase 1
    **then send** $(echo, I_0, ET', 0)$ to all;
  **if received** $(echo, I_0, v\prime, 0)$ messages from $n - f$ distinct nodes by
      the end of phase 2
    **then** invoke **broadcast**$(p, v\prime, 2)$; **stop** and **return**$(v\prime)$.
**round** $r$ **for** $r = 2$ **to** $r = f + 2$ **do:**
  **if** $v \neq 0$ **then** invoke **broadcast**$(p, v, r)$; **stop** and **return**$(v)$.
  by the end of round $r$:
    **if** in rounds $r\prime \leq r$ **accepted** $(I_0, v, 0)$ and $(q_i, v\prime, i)$ for all $i$, $2 \leq i \leq r$,
        where all $q_i$ distinct
      **then** $v := v\prime$;
    **if** $|broadcasters| < r - 1$ **then stop** and **return**$(0)$;
**stop** and **return**$(v)$.
**end**

Nodes stop participating in the Strong-Byz-Consensus protocol when they are instructed to do so. They stop participating in its broadcast primitive by the end of the round in which they stop the Strong-Byz-Consensus. The only exception is when they stop in the 2nd phase of the algorithm. In this special case they stop participation in the broadcast primitive by the end of the 2nd round.

The main feature of the protocol is that when all correct nodes begin with the same value of ET, all stop within 1 round (2 phases). This early stopping

feature brings to a fast convergence during normal operation of the system, even when faulty nodes are present. One can employ standard optimization to save in the number of messages, and to save in a couple of phases.

**Theorem 3.** *The Strong-Byz-Consensus satisfies the following properties:*
**Termination:** *The protocol terminates in a finite time.*
*If the system is in the assumption boundaries, $n > 3f$, and all correct nodes invoke the protocol within $\sigma$ of each other, and messages of correct nodes are received and processed by participating correct nodes then:*
**Agreement:** *The protocol returns the same value at all correct nodes.*
**Validity:** *If all correct nodes invoke the protocol with the same value, then the protocol returns that value, and*
**Early-stopping:** *in such a case all correct nodes stop within 1 round.*

*Proof.* To prove the termination property notice that no matter at what state each node is, very node terminates the protocol within $f+2$ rounds on its clock.

Notice that if correct nodes invoke the protocol within a $\sigma$ of each other and since each phase is long enough to ensure that messages sent at a beginning of a phase by a correct node is received by any other before the end of that phase at the target node. Note that the condition that all nodes process all messages of correct nodes capture the case in which some correct nodes may begin the protocol early and send their messages before others started the protocol. The way we use the protocol we ensure that the buffers of all correct nodes are reset before the first one sends any message related to the current invocation of the protocol.

To prove the validity and the early-stopping properties observe that since $n > 3f$, if all correct nodes invoke the protocol with the same value, then they all have $v \neq 0$ within the first round, and immediately invoke **broadcast** and stop and return the same initial value.

The proof of the agreement property is very similar to the proof of the protocol in [**?**] and will be omitted (see the Appendix for details of the Broadcast primitive). The proof needs to address the possible multi value, by arguing that at most one value can be sent at phases 2 to $f + 2$. Notice that the variations we made do not affect the basic proof. Note that the **broadcast**$(p, v, 2)$ invoked in the second phase of the protocol carries the variable 2 to make the proof conceptually agree with the arguments of the original proof of [**?**]. □

## 6 Analysis and Comparison to other Clock Synchronization Algorithms

Our algorithms require reaching consensus in every cycle. This implies that the cycle should be long enough to allow for the consensus to terminate at all correct nodes. This implies having $cycle \geq 2\sigma + 3(2f + 4)d$, assuming that the consensus algorithm takes $(f + 2)$ rounds of $3d$ each. For simplicity we also assume $M$ to be large enough so that it takes at least a cycle for the clocks to wrap around.

The convergence and closure of Pulse-Clock-Synch and the additional algorithm follows from the self-stabilization of the pulse synchronization procedure

| Algorithm | Self-Stabilizing /Byzantine | Precision $\gamma$ | Accuracy | Convergence Time | Messages |
|---|---|---|---|---|---|
| Pulse-Clock-Synch | SS+BYZ | $5d + O(\rho)$ | $3d + O(\rho)$ | $cycle + 3(2f + 5)d$ | $O(nf^2)$ |
| NOADJUST-CS | SS+BYZ | $2d + O(\rho)$ | $3d + O(\rho)$ | $cycle$ | $O(n^2)$ |
| DHSS [?] | BYZ | $d + O(\rho)$ | $(f + 1)d + O(\rho)$ | $2(f + 1)d$ | $O(n^2)$ |
| LL-APPROX [?] | BYZ | $5\epsilon + O(\rho)$ | $\epsilon + O(\rho)$ | $d + O(\epsilon)$ | $O(n^2)$ |
| DW-SYNCH [?] | SS+BYZ | 0 (global pulse) | 0 (global pulse) | $M2^{2(n-f)}$ | $n^2M2^{2(n-f)}$ |
| DW-BYZ-SS [?] | SS+BYZ | $4(n - f)\epsilon + O(\rho)$ | $(n - f)\epsilon + O(\rho)$ | $O(n)^{O(n)}$ | $O(n)^{O(n)}$ |
| PT-SYNC [?] | SS | 0 (global pulse) | 0 (global pulse) | $4n^2$ | $O(n^2)$ |

**Table 1.** Comparison of Clock Synchronization Algorithms ($\epsilon$ is the uncertainty of the message delay). The convergence time is in pulses for the algorithms utilizing a global pulse system and in network rounds for the other semi-synchronous protocols. PT-SYNC assumes the use of shared memory and thus the "message complexity" is of the "equivalent messages".

and from the self-stabilization and termination of the Byzantine Consensus algorithm.

Note that $\Psi_i$, defined in Section **??**, represents the actual deviation of an individual correct clock ($p_i$,) from a real-time interval. The accuracy of the clocks is the bound on the deviation of correct clocks from a real-time interval. The clocks are repeatedly adjusted in order to minimize the accuracy. Following a synchronization of the clock values, that is targeted to occur once in a cycle, correct clocks can be adjusted by at most $ADJ$, where

$$-2d(1-\rho) - 2\rho\frac{cycle}{1+\rho} \leq ADJ \leq 3d(1+\rho) + 2\rho\frac{cycle}{1-\rho}.$$

Should the initial clock values reflect real-time and their initial states consistent, then this determines the accuracy of the clocks with respect to real-time (and not only in terms of a real-time interval), as long as the system stays within the assumption boundaries and clocks do not wrap around.

The precision, $\gamma$, that is guaranteed should be at least the maximal value derived from Theorem **??**, thus $\gamma \geq 3d(1+\rho) + 2\rho\frac{cycle}{1-\rho} + \sigma(1+\rho) + cycle \cdot 2\rho + \sigma(1+2\rho)2\rho \geq ADJ + \gamma_0 + cycle \cdot 2\rho + \sigma(1+2\rho)2\rho$. A more careful discussion can point out on the overlap of some of the bounds and can reduce the bound on $\gamma$.

The only Byzantine self-stabilizing clock synchronization algorithms, to the best of our knowledge, are published in [?,?]. Two randomized self-stabilizing Byzantine clock synchronization algorithms are presented, designed for fully connected communication graphs, use message passing which allow faulty nodes to send differing values to different nodes, allow transient and permanent faults during convergence and require at least $3f + 1$ processors. The clocks wrap around where $M$ is the upper bound on the clock values held by individual processors. The first algorithm assumes a common global pulse system and synchronizes in expected $M \cdot 2^{2(n-f)}$ global pulses. The second algorithm in [?] does not use a global pulse system and is thus partially synchronous similar to our model. The

convergence time of the latter algorithm is in expected $O((n-f)n^{6(n-f)})$ time. Both algorithms thus have drastically higher convergence times than ours.

In Table 1 we compare the parameters of our protocols to previous classic Byzantine clock synchronization algorithms, to non-Byzantine self-stabilizing clock synchronization algorithms and to the prior Byzantine self-stabilizing clock synchronization algorithms. It shows that our algorithm achieves precision, accuracy, message complexity and convergence time similar to non-stabilizing algorithms, while being self-stabilizing. The $O(nf^2)$ message complexity as well as the convergence time come from the specific Byzantine Consensus algorithm used.

Note that the use of global clock ticks does not make the synchronization problem trivial as the nodes will still miss a common point in time where the new clock value is agreed and the clocks adjusted accordingly (see [**?**]).

Note that if instead of using the pulse synchronization procedure of Section **??**, one uses the pulse synchronization of [**?**] then the precision can somewhat improve, but the cycle, and therefore the convergence time would drastically increase.

## References


1. E. Anceaume, I. Puaut, "*Performance Evaluation of Clock Synchronization Algorithms*", Technical report 3526,INRIA, 1998.
2. A. Arora, S. Dolev, and M.G. Gouda, "*Maintaining digital clocks in step*", Parallel Processing Letters, 1:11-18, 1991.
3. J. Brzeziński, and M. Szychowiak, "*Self-Stabilization in Distributed Systems - a Short Survey*, Foundations of Computing and Decision Sciences, Vol. 25, no. 1, 2000.
4. A. Daliot, D. Dolev and H. Parnas, "*Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks*", Proc. Of the Sixth Symposium on Self-Stabilizing Systems, pp. 32-48, 2003.
5. D. Dolev, J. Halpern, and H. R. Strong, "*On the Possibility and Impossibility of Achieving Clock Synchronization*", J. of Computer and Systems Science, Vol. 32:2, pp. 230-250, 1986.
6. D. Dolev, H. R. Strong, "*Polynomial Algorithms for Multiple Processor Agreement*", In Proceedings, the 14th ACM SIGACT Symposium on Theory of Computing, 401-407, May 1982. (STOC-82)
7. D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, "*Dynamic Fault-Tolerant Clock Synchronization*", J. Assoc. Computing Machinery, Vol. 42, No.1, pp. 143-185, Jan. 1995.
8. S. Dolev, "*Possible and Impossible Self-Stabilizing Digital Clock Synchronization in General Graphs*", Journal of Real-Time Systems, no. 12(1), pp. 95-107, 1997.
9. S. Dolev, "*Self-Stabilization*", The MIT Press, 2000.
10. S. Dolev, and J. L. Welch, "*Self-Stabilizing Clock Synchronization in the presence of Byzantine faults*", Proc. Of the Second Workshop on Self-Stabilizing Systems, pp. 9.1-9.12, 1995.
11. S. Dolev and J. L. Welch, "*Wait-free clock synchronization*", Algorithmica, 18(4):486-511, 1997.
12. M. J. Fischer, N. A. Lynch and M. Merritt, "*Easy impossibility proofs for distributed consensus problems*", Distributed Computing, Vol. 1, pp. 26-39, 1986.

13. T. Herman, "*Phase clocks for transient fault repair*", IEEE Transactions on Parallel and Distributed Systems, 11(10):1048-1057, 2000.
14. B. Liskov, "*Practical Use of Synchronized Clocks in Distributed Systems*", Proceedings of $10^{th}$ ACM Symposium on the Principles of Distributed Computing, 1991, pp. 1-9.
15. B. Patt-Shamir, "*A Theory of Clock Synchronization*", Doctoral thesis, MIT, Oct. 1994.
16. M. Papatriantafilou, P. Tsigas, "*On Self-Stabilizing Wait-Free Clock Synchronization*", Parallel Processing Letters, 7(3), pages 321-328, 1997.
17. F. Schneider, "*Understanding Protocols for Byzantine Clock Synchronization*", Technical Report 87-859, Dept. of Computer Science, Cornell University, Aug. 1987.
18. Sam Toueg, Kenneth J. Perry, T. K. Srikanth, "*Fast Distributed Agreement*", Proceedings, Principles of Distributed Computing, 87-101 (1985).
19. J. L. Welch, and N. Lynch, "*A New Fault-Tolerant Algorithm for Clock Synchronization*", Information and Computation 77, 1-36, 1988.

## Appendix - The Broadcast Primitive

For being self contained we present in this appendix the **broadcast** (and accept) primitive of Toueg, Perry, and Srikanth [**?**] that is used in the Strong-BYZ-Consensus presented above.

In the procedure whenever the nodes are required to consider messages received in a given phase it should be interpreted as a message received by the end of the given phase. The difference comes from the need to use the procedure in an environment that is not tightly synchronous, as the environment we assume in this paper. Note that when a node invokes the procedure it evaluates all the messages in its buffer that are relevant to the procedure.

**Procedure Broadcast**$(p, v, r)$ */* executed per such triple */*
**round** $= k$ :
  **phase** $= 2k - 1$ :
    node $p$ sends $(init, p, m, k)$ to all nodes;
  **phase** $= 2k$ :
    **if received** $(init, p, m, k)$ from $p$ in phase $2k - 1$
      and received only one $(init, p, \_, \_)$ message in all previous phases
    **then send** $(echo, p, m, k)$ to all;
    **if received** $(echo, p, m, k)$ msgs from $\geq n - f$ distinct nodes in phase $2k$
    **then accept**$(p, m, k)$;
**round** $k + 1$ :
  **phase** $= 2k + 1$ :
    **if received** $(echo, p, m, k)$ from $\geq n - 2f$ distinct nodes $q$ in phase $2k$
      and received only one $(echo, p, \_, \_)$ message from each such $q$
    **then send** $(init\prime, p, m, k)$ to all;
    **if received** $(init\prime, p, m, k)$ from $\geq n - 2f$ in phase $2k + 1$
    **then** $broadcasters := broadcasters \bigcup \{p\}$;
  **phase** $= 2k + 2$ :
    **if received** $(init\prime, p, m, k)$ from $\geq n - f$ distinct nodes in phase $2k + 1$
    **then send** $(echo\prime, p, m, k)$ to all;

**if received** $(echo\prime, p, m, k)$ from $\geq n - f$ in phase $2k + 2$
**then accept**$(p, m, k)$;
**round** $r \geq k + 2$ :
**phases** $2r - 1, 2r$ :
**if received** $(echo\prime, p, m, k)$ from $\geq n - 2f$ distinct nodes in previous phases and not sent $(echo\prime, p, m, k)$
**then send** $(echo\prime, p, m, k)$ to all;
**if received** $(echo\prime, p, m, k)$ from $\geq n - f$ distinct nodes in previous phases
**then accept**$(p, m, k)$;
**end**

The Broadcast primitive satisfies the following 4 properties. In [**?**] it was proven that the properties hold under the assumption that $n > 3f$.

1. *(Correctness)* If correct node $p$ broadcasts $(p, m, k)$ in round $k$, then every correct node accept $(p, m, k)$ in the same round.
2. *(Unforgeability)* If correct node $p$ does not broadcast $(p, m, k)$, ten no correct node accepts $(p, m, k)$.
3. *(Relay)* If a correct node accepts $(p, m, k)$ in round $r \geq k$, then every other correct node accepts $(p, m, k)$ in round $r + 1$ or earlier.
4. *(Detection of broadcasters)* If a correct node accepts $(p, m, k)$ in round $k$ or later, then every correct node has $p \in broadcasters$ at the end of round $k + 1$. Furthermore, if correct node $p$ does not broadcast any message, then a correct node can never have $p \in broadcasters$.

One can show that their proofs hold in our environment once the system is back in its assumption boundaries.