Byzantine Self-Stabilizing Pulse in a Bounded-Delay Model*

Danny Dolev** and Ezra N. Hoch

School of Engineering and Computer Science,
The Hebrew University of Jerusalem, Israel,
{dolev,ezraho}@cs.huji.ac.il

Abstract. "Pulse Synchronization" intends to invoke a recurring distributed event at the different nodes, of a distributed system as simultaneously as possible and with a frequency that matches a predetermined regularity. This paper shows how to achieve that goal when the system is facing both transient and permanent (*Byzantine*) failures.
Byzantine nodes might incessantly try to de-synchronize the correct nodes. Transient failures might throw the system into an arbitrary state in which correct nodes have no common notion what-so-ever, such as time or round numbers, and thus cannot use any aspect of their own local states to infer anything about the states of other correct nodes. The algorithm we present here guarantees that eventually all correct nodes will invoke their pulses within a very short time interval of each other and will do so regularly.
The problem of pulse synchronization was recently solved in a system in which there exists an outside beat system that synchronously signals all nodes at once. In this paper we present a solution for a bounded-delay system. When the system in a steady state, a message sent by a correct node arrives and is processed by all correct nodes within a bounded time, say d time units, where at steady state the number of Byzantine nodes, f, should obey the $n > 3f$ inequality, for a network of n nodes.

1 Introduction

When constructing distributed systems, fault tolerance is a major consideration. Will the system fail if part of the memory has been corrupted (e.g. by a buffer overrun)? Will it withstand message losses? Will it overcome network disconnections? To build distributed systems that are fault tolerant to different types of faults, two main paradigms have been used: The Byzantine model and the self-stabilizing model

The *Byzantine* fault paradigm assumes that up to some fraction of the nodes in the system (typically one-third) may behave arbitrarily. Moreover, these nodes can collude in order to try and bring the system down (for more on *Byzantine* faults, see [1]).

* This paper will appear in SSS 2007. Distribution of this paper is prohibited.
** Part of the work was done while the author visited Cornell university. The work was funded in part by ISF, ISOC, NSF, CCR, and AFOSR.

The self-stabilization model assumes that the system might be thrown out of its assumed working conditions for some period of time. Once the system is back to its normal boundaries, all nodes should converge to the desired solution. For example, starting from any memory state, after a finite time, all nodes should have the same clock value (for more on self-stabilization, see [2]).

The strength of self-stabilizing systems emerges from their ability to continue functioning after recovering from a massive disruption of their assumed working conditions. The advantage of *Byzantine* tolerant systems comes from being able to withstand any kind of faults while the system operates in its known boundaries. By combining these two fault models, a distributed system can continue operating properly in the presence of faults as long as "everything is going well"; however, if "things aren't going well", the system will be able to recover once the conditions hold again, and the ratio of *Byzantine* nodes hold.

Clock synchronization is a fundamental building block in many distributed systems; hence, creating a self-stabilizing *Byzantine* tolerant clock synchronization is a desirable goal. Once such an algorithm exists, one can stabilize *Byzantine* tolerant algorithms that were not designed for self-stabilization (see [3]). Clock synchronization can be created upon a PULSEing algorithm (see [4]), which is the main motivation behind the current paper.

The main contribution of the current paper is to develop a pulse synchronization algorithm that converges once the communication network resumes delivering messages within bounded, say d, time units, and the number of Byzantine nodes, f, obeys the $n > 3f$ inequality, for a network of n nodes. The attained pulse synchronization tightness is $3d$ with a deterministic convergence time of a constant number of pulse cycles (each containing $O(f)$ communication rounds).

1.1 Related work

Algorithms combining self-stabilization and *Byzantine* faults, can be divided into two classes. The first consists of problems in which the state of each node is determined locally (see [5,6,7]). The other class contains problems such that a node's state requires global knowledge - for example, clock synchronization such that every two nodes' clocks have a bounded difference that is independent of the diameter of the network (see [8,9,4]). The current paper is of the latter class.

The current paper makes use of the self-stabilizing *Byzantine* agreement algorithm (ss-Byz-Agree) presented in [10]. The above work operates in exactly the same model as the current paper, and its construction will be used as the basic building block in our current solution. Appendix A lists the main properties of this building block.

When discussing clock synchronization, it is common to represent the clocks as an integer value that progresses linearly in time (see [11]). This was previously termed digital clock synchronization ([12,13,14,15]) or "synchronization of phase-clocks" ([16]). In the current paper we provide a PULSEing algorithm; however, when comparing it to other results, we consider the digital clock synchronization algorithm that can be built upon it (as in [4]).

The first ever algorithm to address self-stabilizing *Byzantine* tolerant clock synchronization is presented in [8]. [8] discusses two models; one is synchronous, that is, all nodes are connected to some global "tick" system that produces "ticks" that reach all nodes at the same time, and messages sent at any given tick reach their destination before the following tick. The second model is a bounded-delay network, in which there is no common tick system, but messages have a bounded delay on their delivery time. There is no reason to consider an asynchronous model, since even a single fail-stop failure can't be overcome (see [17]). Note that the bounded-delay model contains the first (synchronous) one. [8] gives two solutions, one for each model, both of which converge in expected exponential time; both algorithms support $f < \frac{n}{3}$.

In [9] clock synchronization is reached in deterministic linear time. However, [9] addresses only the synchronous model, and supports only up to $f < \frac{n}{4}$. In [18], a PULSEing algorithm that operates in the synchronous model is presented, which converges in deterministic linear time, and supports $f < \frac{n}{3}$, matching the lower bounds both in the maximal number of *Byzantine* nodes, and in the convergence time (see [19] for lower bounds). In [20] a very complicated pulse synchronization protocol, in the same model as the current paper, was presented.

The current paper presents a PULSE-synchronization algorithm, which has deterministic linear convergence time, supports $f < \frac{n}{3}$, and operates in a bounded-delay model.

2 Model and Problem Definition

The model used in this paper consists of n nodes that can communicate via message passing. Each message has a bounded delivery time, and a bounded processing time at each node; in addition the message sender's identity can be validated. The network is not required to support broadcast.

Each node has a local clock. Local clocks might show different readings at different nodes, but all clocks advance at approximately the real-time rate.

Nodes may be subject to transient faults, and at any time a constant fraction of nodes may be *Byzantine*, where f, the number of *Byzantine* nodes satisfies $f < \frac{n}{3}$.

Definition 1. *A node is* **non-faulty** *if it follows its algorithm, processes messages in no more than π time units and has a bounded drift on its internal clock. A node that is not* **non-faulty** *is considered* **faulty** *(or Byzantine). A node is* **correct** *if it has been* **non-faulty** *for Δ_{node} time units.*[1]

Definition 2. *A communication network is* **non-faulty** *if messages arrive at their destinations within δ time units, and the content of the message, as well as the identity of the sender, arrive intact. A communication network is* **correct** *if it has been* **non-faulty** *for Δ_{net} time units.*[2]

[1] The value of Δ_{node} will be stated later.
[2] The value of Δ_{net} is stated below.

The value of Δ_{net} is chosen so if at time t_1 the communication network is non-faulty and stays so until $t_1 + \Delta_{net}$, then only messages sent *after* t_1 are received by non-faulty nodes.

Definition 3. *A system is* **coherent** *if the network is* **correct** *and there are at least* $n - f$ **correct nodes**.

Once the system is coherent, a message between two correct nodes is sent, received and processed within d time units, where d is the sum of δ, π and the upper bound on the potential drift of correct local timers during such a period. Δ_{net} should be chosen in such a way as to satisfy $\Delta_{net} \geq d$.. Since d includes the drift factor, and since all the intervals of time will be represented as a function of d, we will not explicitly refer to the drift factors in the rest of the paper.

2.1 Self-stabilizing Byzantine Pulse-Synchronization

Intuitively, the PULSE synchronization problem consists of synchronizing the correct nodes so they invoke their pulses together *Cycle* time apart. That is, all correct nodes should invoke pulses within a short interval, then not invoke a pulse for approximately *Cycle* time, then invoke pulses again within a short interval, and so on. Adding "Self-stabilizing *Byzantine*" to the PULSE synchronization problem, means that starting from any memory state and in spite of ongoing *Byzantine* faults, the correct nodes should eventually invoke pulses together *Cycle* time apart.

Since message transmission time varies and also due to the *Byzantine* presence, one cannot require the correct nodes to invoke pulses exactly *Cycle* time apart. Instead, $cycle_{min}$ and $cycle_{max}$ are values that define the bounds on the actual CYCLE length in a correct behavior. The protocol presented in this paper achieves $cycle_{min} = Cycle \leq \text{CYCLE} \leq Cycle + 12d = cycle_{max}$.

To formally define the PULSE synchronization problem, a notion of "PULSEing together" needs to be addressed.

Definition 4. *A correct node p invokes a pulse* **near** *time unit t if it invokes a pulse in the time interval $[t - \frac{3}{2} \cdot d, t + \frac{3}{2} \cdot d]$. Time unit t is a* **pulsing point** *if every correct node invokes a pulse near t.*

Definition 5. *A system is in a* **synchronized_pulsing_state** *in the time interval $[r_1, r_2]$ if*

1. *there is some pulsing point $t_0 \in [r_1, r_1 + cycle_{\max}]$;*
2. *for every pulsing point $t_i \leq r_2 - cycle_{\max}$ there is another pulsing point t_{i+1}, $t_{i+1} \in [t_i + cycle_{\min}, t_i + cycle_{\max}]$;*
3. *for any other pulsing point $\bar{t} \in [r_1, r_2]$, there exists i, such that $|t_i - \bar{t}| \leq \frac{3}{2} \cdot d$.*

Intuitively, the above definition says that in the interval $[r_1, r_2]$ there are pulsing points that are spaced at least $cycle_{min}$ apart and no more than $cycle_{max}$ apart.

Definition 6. *Given a coherent system,* The Self-Stabilizing Pulse Synchronization Problem *requires that:*

Convergence: *Starting from an arbitrary system state, the system reaches a synchronized_pulsing_state within a finite amount of time.*

Closure: *If the system is in a synchronized_pulsing_state in some interval* $[t_1, t_2]$ *(s.t.* $t_2 > t_1 + cycle_{max}$*), then it is also in a synchronized_pulsing_state in the interval* $[t_1, t]$ *for any* $t > t_2$*.*

3 Solution Overview

The main algorithm, ERRATIC-PULSER, assumes a self-stabilizing, *Byzantine* tolerant, distributed agreement primitive, $\mathcal{Q}$, which is defined in the following section. A protocol providing the requirements of $\mathcal{Q}$ is presented in Section 5.

Using $\mathcal{Q}$, the ERRATIC-PULSER algorithm produces agreement among the correct nodes on different points in time at which they invoke pulses; and these points become sparse enough. Using this basic point-in-time agreement, a full PULSE algorithm is built, named BALANCED-PULSER. By using the basic PULSEing pattern produced by ERRATIC-PULSER, BALANCED-PULSER manages to solve the PULSE-synchronization problem.

During the rest of this paper, the constants *Cycle*, $cycle_{min}$ and $cycle_{max}$ are used freely. However, it is important to note that *Cycle* must be chosen such that it is large enough. The exact limitations on the possible values of *Cycle* will be stated later. An explanation on how to create a PULSEing algorithm with an arbitrary *Cycle* value is presented in Section 9.

4 The $\mathcal{Q}$ Primitive

$\mathcal{Q}$ is a primitive executed by all the nodes in the system. However, each invocation of a specific $\mathcal{Q}$ is associated with some node, p, hence a specific invocation will sometimes be referred to as $\mathcal{Q}(p)$. That is, $\mathcal{Q}(p)$ is a distributed algorithm, executed by all nodes, and triggered by p (p's special role in the execution of $\mathcal{Q}(p)$ will be elaborated upon later). In the following discussion several instances of $\mathcal{Q}(p)$ may coexist, but it will be clear from the context to which instance $\mathcal{Q}(p)$ refers. Each instance is a separate copy of the protocol and each node executes each instance separately.

$\mathcal{Q}(p)$ is a "consensus primitive", that is, each node q has an input value v_q, and upon completing the execution of $\mathcal{Q}(p)$ it produces some output value V_q. The input values and output values are boolean, i.e., $v_q, V_q \in \{0, 1\}$. Denote by τ_q the local-time at node q at which V_q is defined; that is τ_q is the local time at node q at which q determines the value of V_q (and terminates $\mathcal{Q}(p)$).

The un-synchronized and distributed nature of $\mathcal{Q}(p)$ requires distinguishing between two stages. The first stage is when p attempts to invoke $\mathcal{Q}(p)$; this attempted invocation involves exchanging messages among the nodes. The second stage is when enough correct nodes agree to join p's invocation of $\mathcal{Q}(p)$, and

hence start executing $\mathcal{Q}(p)$. When p is correct, the first stage and the second stage are close to each other; however, when p is faulty, no a priori bound can be set on the time difference between the first and the second stages. Note that p itself joins the instance of $\mathcal{Q}(p)$ only after the preliminary invocation stage.

Informally, $join_q$ is the time at which q agrees to join the instance of $\mathcal{Q}(p)$ (which is also the time at which q determines its input value v_q.) Following this stage it actively participates in determining the output value of $\mathcal{Q}(p)$. The implementation of $\mathcal{Q}(p)$ needs to explicitly instruct a node when to determine its input value.

In the following discussion, rt_{invoke} will denote the time at which p invoked $\mathcal{Q}(p)$ and $join_{first}$ will denote the time value at which the first correct node joins the execution of $\mathcal{Q}(p)$; $join_{last}$ will denote the time value at which the last correct node joins p in executing $\mathcal{Q}(p)$. That is, $join_{first} = \min_{\text{correct } q}\{join_q\}$ and $join_{last} = \max_{\text{correct } q}\{join_q\}$.

$\mathcal{Q}(p)$ is self-stabilizing, and its properties hold once the system executing it is coherent for at least $\Delta_{\mathcal{Q}}$ time. In other words, no matter what the initial values in the nodes' memory may be, after the system has been coherent for $\Delta_{\mathcal{Q}}$ time, the properties of $\mathcal{Q}(p)$ will hold.[3]

$\mathcal{Q}(p)$'s properties follow. Observe that there are different requirements, depending on whether p is a correct node or not.

1. For any node p invoking $\mathcal{Q}(p)$, the following holds:
   (a) *Agreement:* all correct nodes that have terminated have the same output value. That is, for any pair of correct nodes, q and q', which have completed $\mathcal{Q}(p)$, $V_q = V_{q'}$. V denotes this common output value.
   (b) *Validity:* if all correct nodes have the same input value ν then $V = \nu$.
   (c) *Termination:* if some correct node joins $\mathcal{Q}(p)$ then all correct nodes terminate within Δ_{max} time units from $join_{first}$ but no quicker than Δ_{min}. That is, for a correct q, $rt(\tau_q) \in [join_{first} + \Delta_{min}, join_{first} + \Delta_{max}]$, where τ_q is the local time at which q determines the value of V_q, and $rt(\tau_q)$ is the time at which this takes place.
   (d) *Tightness:* if a correct node terminates, then for any correct nodes q, q': $|rt(\tau_q) - rt(\tau_{q'})| \leq 3 \cdot d$.
   (e) *Collaboration:* if one correct node joins the execution of $\mathcal{Q}(p)$, then all correct nodes join the execution of $\mathcal{Q}(p)$ within $3 \cdot d$ of each other; that is, $|join_{last} - join_{first}| \leq 3 \cdot d$.
2. For a correct node p, starting the execution of $\mathcal{Q}(p)$ at time rt_{invoke}, the following holds:
   (a) *Strong Termination:* $join_{first} \leq rt_{invoke} + 3 \cdot d$. That is, the first correct node to join p in executing $\mathcal{Q}(p)$ does so within $3 \cdot d$ time from p's invocation of $\mathcal{Q}(p)$. Combined with *termination*, this property means that all correct nodes terminate by $rt_{invoke} + 3 \cdot d + \Delta_{max}$.
   (b) *Separation:* p does not start $\mathcal{Q}(p)$ more than once every $3 \cdot \Delta_{max}$ time units.

[3] $\Delta_{\mathcal{Q}}$ is defined below.

3. The following holds for a faulty p, invoking $\mathcal{Q}(p)$:
   (a) *Separation:* if a correct node q assigns an output value for $\mathcal{Q}(p)$ at some time t_1, then it does not assign an output value for $\mathcal{Q}(p)$ again before $t_1 + 2 \cdot \Delta_{min}$.

Remark 1. According to "termination" if $join_{first}$ is not defined, all correct nodes do not terminate. This implies that all correct nodes terminate if and only if some correct node joins p in executing $\mathcal{Q}(p)$.

Note that p may require the invocation of several $\mathcal{Q}(p)$ instances concurrently. To differentiate between these instances, they are marked with an additional index, e.g $\mathcal{Q}_1(p)$, $\mathcal{Q}_2(p)$, etc. Each such instance has its own memory space, and hence is independent of other instances. According to the *separation* property, a correct node does not execute the same instance of $\mathcal{Q}(p)$ too often. That is, $\mathcal{Q}_1(p)$ is not executed until the previous $\mathcal{Q}_1(p)$ has terminated. A faulty node p may try to invoke $\mathcal{Q}_1(p)$ as often as it likes, however correct nodes will ignore the multiple executions.

5 Implementing $\mathcal{Q}(p)$, the ss-Byz-Q Algorithm

The implementation of $\mathcal{Q}(p)$ makes use of ss-Byz-Agree ([10]). The properties of ss-Byz-Agree and its guarantees are listed in Appendix A. In ss-Byz-Agree, when a node p wants to start an agreement on some value, it sends (*Initiator*, p, v_p) to all other nodes. Nodes receiving this message, initiate the ss-Byz-Agree algorithm, and start participating in the agreement. Other nodes, that have not received the (*Initiator*, p, v_p) message (in case p is *Byzantine*), join the ss-Byz-Agree algorithm once they are "convinced" that enough correct nodes are already executing ss-Byz-Agree on p's value.

This leads to the following insight. If a correct node q ignores an (*Initiator*, p, v_p) message sent by a *Byzantine* node (for any reason), it does not change the properties of ss-Byz-Agree. Since due to p's *Byzantine* nature, if p would have not sent this specific message to q, ss-Byz-Agree's properties would still hold. Hence, whether p sends the message and a correct node ignores it, or p doesn't send the message at all, the properties of ss-Byz-Agree remain the same. Note that this is true only if p is *Byzantine*. In what follows, when a node *rejects* a message it ignores it, and when it *accepts* a message it continues to execute the protocol as instructed.

Figure 1 presents an algorithm that implements the $\mathcal{Q}$ primitive. If node p wants to invoke $\mathcal{Q}(p)$, it does so by executing ss-Byz-Agree (p, *start_Q*) (this is the *Init* stage), which means it sends (*Initiator*, p, *start_Q*) messages to other nodes. This action triggers the *prolog* stage of the protocol. If this stage completes successfully, each correct node peforms a timing test to determine whether to join the computation of the primitive $\mathcal{Q}(p)$. The algorithm is executed in the background continuously, and it responds to messages / events that are triggered by p's execution of ss-Byz-Agree (p, *start_Q*).

Algorithm ss-Byz-Q
(implementing $\mathcal{Q}(p)$) /* executed at node q */

Init: **If** $p = q$ invoke ss-Byz-Agree $(p, start\_Q)$);
/* by sending (Initiator, p, $start\_Q$)) message to all */

Prolog: **On** receiving (*Initiator*, p, $start\_Q$) message from p
if $local_q > last_q[p] + 2 \cdot \Delta_{max}$ **then** accept the message;
else ignore the message;

The Primitive $\mathcal{Q}(p)$:

1. **On** returning from ss-Byz-Agree for p with value "$start\_Q$" **do**
   if $local_q > last_q[p] + \Delta_{max} + 3 \cdot d$ **then**
   begin
   determine the input value v_q; /* this is when q joins $\mathcal{Q}(p)$ */
   $start_q[p] := local_q$;
   reset $val_q[p, \_]$;
   wait for $3 \cdot d$ and then invoke ss-Byz-Agree $(q, (p, v_q))$;
   /* by sending (Initiator, q, (p, v_q)) message to all */
   end
   $last_q[p] := local_q$;
2. **On** receiving (*Initiator*, p', $(p, v_{p'})$)
   if $local_q \leq start_q[p] + 7 \cdot d$ **then** accept the message;
   else ignore the message;
3. **On** returning from ss-Byz-Agree for p' with value $(p, v_{p'})$ **do**
   $val_q[p, p'] = v_{p'}$;
4. **At** time $local_q = start_q[p] + \Delta + 17 \cdot d$
   for all p', check the agreement values $val_q[p, p']$
   if there are $n - f$ 1's, then set $V_q[p] := 1$, otherwise set $V_q[p] := 0$;
   return $V_q[p]$ as $\mathcal{Q}(p)$'s output value;

Cleanup:
for any p: if $last_q[p] > local_q$ then $last_q[p] := local_q$
for any p: if $start_q[p] > local_q$ then $start_q[p] := local_q$

Fig. 1. An algorithm that implements $\mathcal{Q}(p)$

The values of the constants for the ss-Byz-Q algorithm are: $\Delta_{max} := \Delta + 20 \cdot d$ and $\Delta_{min} := \Delta_{max} - 3 \cdot d$, where Δ represents the maximal time required to complete ss-Byz-Agree ($\Delta := 7(2f + 3)d$, see Appendix A).

In the $\mathcal{Q}(p)$ protocol in Figure 1, $local_q$ represents the local time at each node q; in addition there are two arrays of values: $start_q, last_q$. These arrays hold local-time values (per node p) of events regarding $\mathcal{Q}(p)$'s execution at q. $last_q[p]$ is used to ensure that q doesn't participate in $\mathcal{Q}(p)$ too often. $start_q[p]$ is used so that all correct nodes know when to stop collecting values of other nodes (regarding $\mathcal{Q}(p)$'s instance); these values are stored in $val_q[p, p']$.

Remark 2. In the protocols, all the comparisons of the value of $local_q$ to some other value, always compare values that are at most some bounded range apart, say D. To deal with the possible wraparound of the counter $local_q$, it is enough

that the range of values of $local_q$ will be $D' > 2D$. The "cleanup" stage of the protocol (See Figure 1) ensures that comparisons over a circle of size D' are uniquely determined.

Note that the protocol parameters n, f and *Cycle* (as well as the system characteristic d) are fixed constants and thus considered part of the incorruptible correct code.[4] Thus we assume that non-faulty nodes do not hold arbitrary values of these constants.

The value of Δ_{node} is crucial for the following claims. Δ_{node} is used to ensure that non-faulty nodes "run" for some time before they become correct. In the context of this paper, a non-faulty node should not be considered correct when it executes ss-Byz-Q that it might have joined before it was non-faulty. Moreover, since ss-Byz-Q uses $ss-$ Byz-Agree which has its own requirements for a node's correctness, we set $\Delta_{node} := \Delta_{\text{node-ss-byz-agree}} + \Delta_{max} + 3 \cdot d$.[5]

Lemma 1. *Once the system is coherent, if all correct nodes pass the condition in Line 1 during a time interval* $[t_1, t_2]$ *s.t.*
1. $t_2 - t_1 \leq 3 \cdot d$*, and*
2. at t_1 *for any correct node* q *it holds that* $local_q > start_q[p] + \Delta_{max}$*,*
then Agreement, Validity, Termination, Tightness *and* Collaboration *hold.*

Proof. First we show that no $ss-$ Byz-Agree$(p', (p, v_{p'}))$ that was initiated before t_1, terminates after t_1. By assumption, at time t_1, each correct node q has $local_q > start_q[p] + \Delta_{max}$, which means that no correct node has accepted (*Initiator*, p', $(p, v_{p'})$) in the time interval $[t_1 - \Delta_{max} + 7 \cdot d, t_1]$. In the protocol, any correct node that accepts (*Initiator*, p', $(p, v_{p'})$) before $t_1 - \Delta_{max} + 7 \cdot d$, must have terminated the ss-Byz-Agree no later than $t_1 - \Delta_{max} + 7 \cdot d + \Delta + 7 \cdot d$, and hence all correct nodes must have terminated the ss-Byz-Agree no later than $t_1 - \Delta_{max} + 17 \cdot d + \Delta$. Since $\Delta_{max} := \Delta + 20 \cdot d$, we conclude that any ss-Byz-Agree that was invoked before t_1 terminated before t_1.

In addition, due to setting of $last_q[p]$ in Line 1, no correct node will pass the condition in Line 1 again, before $t_1 + \Delta_{max} + 3 \cdot d$. Hence, during the interval $[t_2, t_2 + \Delta_{max}]$ no correct node passes the condition in Line 1. Note that each correct node passes the condition in Line 1 exactly once in the interval $[t_1, t_2]$. Hence, all correct nodes reset $val_q[p, \_]$ in the interval $[t_1, t_2]$ and never do so again before $t_2 + \Delta_{max}$. In a sense, the above means that all correct nodes join p in the interval $[t_1, t_2]$ and do not join p again, until after time $t_2 + \Delta_{max}$.

From the lemma's condition; for any two correct nodes q, q', it holds that $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$. At this stage, each correct node joins p's execution of $\mathcal{Q}(p)$ and hence *Collaboration* holds.

For any pair of correct nodes, q, q', $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$. Moreover, q sends its (*Initiator*, q, (p, v_q)) message $3 \cdot d$ after its $start_q[p]$. Since $|rt(start_q[p]) - rt(start_{q'}[p])| \leq 3 \cdot d$, q' has already set its $start_{q'}[p]$ value when it receives q's (*Initiator*, q, (p, v_q)) message. Similarly, q' receives q's (*Initiator*, q, (p, v_q)) message within $7 \cdot d$ of $start_{q'}[p]$ ($3d$ is the waiting of $3d$ in Line 1 of the

[4] A system cannot self-stabilize if the entire code space can be perturbed, see [21].
[5] $\Delta_{\text{node-ss-byz-agree}} := 14(2 \cdot f + 3) \cdot d + 10 \cdot d$ (see [10]).

protocol, additional $3d$ is the time difference in $start_q$, and d is the uncertainty in message delivery), and thus does not ignore it.

This last argument implies that for every correct node q, any other correct node q' accepts its (*Initiator*, q, (p, v_q)) message, and hence finishes ss-Byz-Agree (q, v_q) before time $start_{q'} + \Delta + 7 \cdot d$. Therefore, every correct node "hears" every other correct node's value. That is, for any triplet of correct nodes, q, q', q'' it holds that $val_q[p, q''] = val_{q'}[p, q'']$.

Consider a *Byzantine* node q. If some correct node q' has accepted its (*Initiator*, q, (p, v_q)) message, then according to point 3 of the "*Timeliness-Agreement*" property (see Appendix A), ss-Byz-Agree will terminate within $\Delta + 7 \cdot d$ time units. Hence, any other correct node q'' will terminate within $3 \cdot d$. Since $|start_{q'} - start_{q''}| \leq 3 \cdot d$, node q'' will have accepted the same value no later than $start_{q''} + \Delta + 16 \cdot d$ ($7d$ come from above, $3d$ come from the difference in $start_q$, $3d$ come from the difference in the termination of ss-Byz-Agree and another $3d$ from the waiting after setting $start_q[p]$; all together $7d + 3d + 3d + 3d = 16d$). Note that this proof holds even though correct nodes may ignore an (*Initiator*, q, (p, v_q)) message sent by a *Byzantine* node q. Since no ss-Byz-Agree that was invoked before t_1 is accepted after t_1, it holds that $val_{q'}[p, q] = val_{q''}[p, q]$. As a result all correct nodes have the same set of values when they consider the output value $v_q[p]$, hence they all agree on the same output value. In addition, if all correct nodes started with "0", they will see at most f "1"s, and hence decide $V = 0$. Moreover, if all correct nodes started with "1", then all correct nodes will decide 1. Thus *Agreement* and *Validity* hold.

Each correct node terminates within $\Delta + 17 \cdot d$ of returning from p's invocation of ss-Byz-Agree (which is the joining point of each correct node to $\mathcal{Q}(p)$), and they all terminate within $3 \cdot d$ time units of each other (since $|rt(start_q) - rt(start_{q'})| \leq 3 \cdot d$). Hence *Termination* and *Tightness* hold. □

The following shows that ss-Byz-Q converges in $\Delta_{\mathcal{Q}} := 4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$. For ss-Byz-Q to operate correctly, ss-Byz-Agree must converge as well. Hence, in the following, we will assume that $\Delta_{ss-\text{Byz-Agree}}$ time has already passed. [6]

Lemma 2. *Once the system has been coherent for $4 \cdot \Delta_{max}$ time units, then for a faulty p, the properties of $\mathcal{Q}(p)$ hold for* ss-Byz-Q.

Proof. Note that once the system is coherent, $start_q[p], last_q[p] \leq local_q$. Notice that $start_q[p]$ can only be updated at Line 1.

Consider the first $2 \cdot \Delta_{max}$ time units following the time at which the system became coherent. If some correct node terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ during this period, then all correct nodes do so within $3 \cdot d$ of each other. Hence, they all set their $last_q[p]$ variable within $3 \cdot d$ time units of each other. That is, the values $rt(last_q[p])$ are at most $3 \cdot d$ units apart from each other. If no correct node terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ for $2 \cdot \Delta_{max}$, then all $last_q[p]$ haven't been updated for $2 \cdot \Delta_{max}$ and hence all $last_q[p] + 2 \cdot \Delta_{max} < local_q$ for every correct node q.

[6] $\Delta_{ss-\text{Byz-Agree}} := 2\Delta + 10d$ (see [10]).

Thus, we conclude that after $2 \cdot \Delta_{max}$ time units either all correct nodes have $rt(last_q[p])$ within $3 \cdot d$ of each other or all correct nodes have $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Note that this state continues to hold as long as no correct node enters Line 1 since $last_q[p]$ is not updated at any correct node. If some correct node does update $last_q[p]$ at Line 1, then all correct nodes do so $3 \cdot d$ time units apart.

Now consider the period between $2 \cdot \Delta_{max}$ and $4 \cdot \Delta_{max}$ time units following the time the system became coherent. If no correct node terminated ss-Byz-Agree $(p, start\_\mathcal{Q})$, then each correct node, q, has $local_q > start_q[p] + \Delta_{max}$ (since $start_q[p]$ had not been updated for at least $2 \cdot \Delta_{max}$ time units). Otherwise, if some correct node q' has terminated ss-Byz-Agree $(p, start\_\mathcal{Q})$, it means that there exists some correct node $\bar{q}$ that accepted $(Initiator, p, start\_\mathcal{Q})$ at the *Prolog* stage. Thus, $local_{\bar{q}} > last_{\bar{q}}[p] + 2 \cdot \Delta_{max}$, which means that until $last_{\bar{q}}[p]$ is reset, $local_{\bar{q}} > last_{\bar{q}}[p] + \Delta_{max} + 3 \cdot d$. Remember that either the $rt(last_q[p])$ of each correct node $\bar{q}$ is within $3 \cdot d$ time units of all other correct nodes, or each correct node has $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Therefore, we have that for all correct nodes, until $last_q[p]$ is reset, $local_q > last_q[p] + \Delta_{max} + 3 \cdot d$; which means that when q terminates ss-Byz-Agree $(p, start\_\mathcal{Q})$ it passes the condition of Line 1, along with all other correct nodes (within a $3 \cdot d$ interval). Therefore, the $rt(start_q[p])$ of all correct nodes are within $3 \cdot d$ time units.

Thus, after $4 \cdot \Delta_{max}$, the $rt(start_q[p])$ of all correct nodes are within $3 \cdot d$ time units *and* $rt(last_q[p])$ within $3 \cdot d$ time units of each other or all correct nodes have $last_q[p] + 2 \cdot \Delta_{max} < local_q$. Note that the next time p invokes ss-Byz-Agree, all correct nodes values of $start_q[p]$ will be greater than their $local_q$ by at least $2 \cdot \Delta_{max}$. Hence, if p invokes ss-Byz-Agree and some correct node terminates that instance of ss-Byz-Agree, then all correct nodes pass the condition of Line 1 within $3 \cdot d$ of each other, and each correct node has $local_q > start_q[p] + \Delta_{max}$. Hence, by Lemma 1 all properties except for *Separation* hold.

To show that *Separation* holds, notice that once a correct node has passed Line 1, it won't do so again for at least $2 \cdot \Delta_{max} - 3 \cdot d$ time units. In addition, it will terminate the current instance of $\mathcal{Q}$ within Δ_{max}. Hence, the next invocation of $\mathcal{Q}$ cannot terminate before $2 \cdot \Delta_{min}$. And *Separation* holds. □

Lemma 3. *Once the system has been coherent for $4 \cdot \Delta_{max}$ time units, then for a correct p, the properties of $\mathcal{Q}(p)$ hold for* ss-Byz-Q*, given that p does not initiate ss-*Byz-Agree *$(p, start\_Q)$ earlier than $3 \cdot \Delta_{max}$ time units following its previous invocation.*

Proof. Since *Agreement, Validity, Termination, Tightness* and *Collaboration* were proven to hold even if p is faulty (under the lemma's conditions), they clearly hold if p is correct. Hence, we still need to prove *Strong Termination* and *Separation*. To prove *Strong Termination*, note that if p is correct, and it has not invoked $\mathcal{Q}(p)$ for $3 \cdot \Delta_{max}$ time units, then when it does invoke $\mathcal{Q}(p)$, all correct nodes will accept the message $(Initiator, p, start\_\mathcal{Q})$ and hence, according to item 2 of the "*Timeliness-Agreement*" property of ss-Byz-Agree (see Appendix A), all correct nodes will terminate within $3 \cdot d$ time units following p's invocation of ss-Byz-Agree $(p, start\_\mathcal{Q})$ and join the execution of $\mathcal{Q}(p)$. *Separation* follows from the conditions of the lemma. □

From the above lemmas, we conclude that after $4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$ time units, ss-Byz-Q behaves according to $\mathcal{Q}$'s properties. Setting $\Delta_{\mathcal{Q}} := 4 \cdot \Delta_{max} + \Delta_{ss-\text{Byz-Agree}}$ satisfies the claim that if the system has been coherent for $\Delta_{\mathcal{Q}}$ time units, then the properties of $\mathcal{Q}$ hold.

Since ss-Byz-Q implements $\mathcal{Q}$'s properties correctly, in the rest of the paper we will use ss-Byz-Q and $\mathcal{Q}$ interchangeably.

6 Constructing the Erratic-Pulser Algorithm

The Erratic-Pulser algorithm (Figure 2) is written in an event-driven fashion; that is, it is continuously executed in the background and no explicit initialization is needed. The algorithm requires invoking two $\mathcal{Q}$ instances per node ($\mathcal{Q}_{start}$ and $\mathcal{Q}_{end}$). In addition, each node has three timers TIMER_{start}, TIMER_{end} and TIMER_{main} with elapsed time of CYCLE_{start}, CYCLE_{end} and CYCLE_{main}, respectively. When TIMER_{start} or TIMER_{end} elapse, an instance of ss-Byz-Q is invoked ($\mathcal{Q}_{start}$ for TIMER_{start} and $\mathcal{Q}_{end}$ for TIMER_{end}). TIMER_{main} is used to determine the value of $WantToPulse$, which is used as the input value for $\mathcal{Q}_{start}$ and $\mathcal{Q}_{end}$. If TIMER_{main} is elapsed, then $WantToPulse := 1$, and once TIMER_{main} is reset, $WantToPulse := 0$ until it elapses again.

Algorithm **Erratic-Pulser** */* executed at node p */*
/ the Qs are executed in the background */*
/ the input value v_q for each $\mathcal{Q}$ instance, is the value of $WantToPulse$ at the time q joins $\mathcal{Q}$ */*

1. **when** TIMER_{start} elapses
   reset TIMER_{start} with CYCLE_{large};
   reset TIMER_{end};
   invoke $\mathcal{Q}_{start}(p)$;
2. **when** TIMER_{end} elapses
   reset TIMER_{end} with CYCLE_{large};
   invoke $\mathcal{Q}_{end}(p)$;
3. $WantToPulse := 1$ if TIMER_{main} has elapsed, and $WantToPulse := 0$, otherwise;
4. **on** returning from either $\mathcal{Q}_{start}(q)$ or $Q_{end}(q)$ for some q with value $V = 1$
   (a) invoke a pulse;
   (b) reset TIMER_{main};
   (c) reset TIMER_{start};

cleanup:
if a TIMER is set with invalid value (below 0 or above its maximal value), reset it; for TIMER_{main}, 0 is a valid value;

Fig. 2. An algorithm achieving basic synchronized PULSEing

The intuition behind the algorithm is that $WantToPulse$ determines when p is willing to invoke a pulse. Once all correct nodes have $WantToPulse = 1$, the next time a ss-Byz-Q instance is invoked, all of them will invoke pulses.
Remark: Notice that there is a difference between TIMER_{start}, TIMER_{end} and TIMER_{main}. TIMER_{start}, TIMER_{end} are timers that when they elapse, an event

occurs, and the algorithm performs some action. These timers are always set, that is, once they elapse, they are reset immediately. TIMER_{main}, on the other hand, will remain in its elapsed state until it is reset. That is, Line 3 is not executed only when TIMER_{main} elapses, but rather it is executed continuously. In a sense, when q wants to read its $WantToPulse$ variable value, it checks whether TIMER_{main} has elapsed; if so then it considers $WantToPulse = 1$, otherwise it reads $WantToPulse = 0$.

The following are the values of the constants used in Erratic-Pulser.

$\text{CYCLE}_{start} := \text{CYCLE}_{main} := Cycle - \Delta_{max} - \Delta_{min}$;

$\text{CYCLE}_{end} = \Delta_{min} - 10 \cdot d$;

$\text{CYCLE}_{large} := 2 \cdot (\Delta_{max} + \text{CYCLE}_{start} + \text{CYCLE}_{end})$.

Note: CYCLE_{main} needs to be larger than $\Delta_{max} + 9 \cdot d$ time units, hence, $Cycle$ must be larger than $2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$ time units.

7 Erratic-Pulser's Correctness Proofs

Definition 7. *A correct node p **pulses-in-unison**, there is a pulsing point t, such that p invokes a pulse near t each time that p invokes a pulse. The system **pulses-in-unison**, if for every correct node p , p **pulses-in-unison***

Remark 3. The definition of "near t" implies that if p **pulses-in-unison** then each time p invokes a pulse there is a time interval $[t_1, t_2]$ such that $|t_2 - t_1| \leq 3 \cdot d$ and each correct node (including p) invokes a pulse within this interval. This also implies that if there exists a correct node p that **pulses-in-unison** then the system **pulses-in-unison**.

Lemma 4. *Once the system has been coherent for $\Delta_{\mathcal{Q}}$ time, the system pulses-in-unison.*

Proof. According to Lemma 2 and Lemma 3 (in Section 5), once the system has been coherent for $\Delta_{\mathcal{Q}}$time units, all copies of ss-Byz-Q behave according to the requirements of $\mathcal{Q}$. This means that all correct nodes see the same output values. Since a correct node invokes a pulse only in accordance with the output of a $\mathcal{Q}$, if some correct node invokes a pulse, then within $3 \cdot d$ time units from its pulse, all correct nodes will also invoke pulses. This means that every correct node pulses-in-unison, which means that the system pulses-in-unison. □

The following lemma proves that a correct node will eventually invoke a pulse. The previous lemma claims that after some time, **if** a correct node invokes a pulse, then all the correct nodes invoke pulses.

Lemma 5. *Eventually some correct node will invoke a pulse. This happens no later than $\Delta_{\mathcal{Q}} + \Delta_{max} + \text{CYCLE}_{large} + \text{CYCLE}_{main} + 3 \cdot d$ time units after the point at which the system becomes coherent.*

Proof. Consider the system $\Delta_{\mathcal{Q}}$ after it becomes coherent: If a correct node invokes a pulse, the lemma holds. Otherwise, after CYCLE_{main} time units, all correct nodes will have $WantToPulse$ as 1. Eventually, after no more than CYCLE_{large},

TIMER_{start} at some correct p will expire, which will initiate $\mathcal{Q}_{start}(p)$, that terminates no more than $\Delta_{max} + 3 \cdot d$ time units afterwards (by *strong termination*), and will have the output value $V = 1$ (since all correct nodes had the input value of $v = 1$). By line 4, of the ERRATIC-PULSER, p will invoke a pulse. □

Lemma 6. *Once the system pulses-in-unison, let t_1 be a time unit at which a correct node p invokes a pulse. Let t_2 be the last time at which p invokes a pulse in the interval $[t_1, t_1 + \Delta_{max} + 3 \cdot d]$. p does not invoke a pulse in the interval $[t_2, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$. p invokes a pulse at some time t_3, where $t_3 \leq t_2 + \text{CYCLE}_{main} + 6 \cdot d + \Delta_{max}$.*

Proof. According to the lemma's assumption the system pulses-in-unison. Hence, when p invokes a pulse at t_1 all correct nodes invoke pulses before time $t_1 + 3 \cdot d$. Define $[t_s, t_e]$ to be the time interval in which all correct nodes have invoked a pulse, such that $t_1 \in [t_s, t_e]$ and $t_e - t_s \leq 3 \cdot d$. All correct nodes execute lines 4.a, 4.b and 4.c during the interval $[t_s, t_e]$. Therefore, the correct nodes' timers $\text{TIMER}_{main}, \text{TIMER}_{start}$ are reset. Hence, after t_e all correct nodes' values of $WantToPulse$ are 0, and hence any correct node that joins any $\mathcal{Q}$ instance after t_e has an input value $v_q = 0$. This holds until TIMER_{main} elapses at some correct node, that is until $t_s + \text{CYCLE}_{main}$. In other words, no correct node joins any $\mathcal{Q}$ instance in the interval $[t_e, t_s + \text{CYCLE}_{main}]$ with input value of 1.

By definition, t_2 is the last time that p invoked a pulse in the interval $[t_1, t_1 + \Delta_{max} + 3 \cdot d]$. Hence, after $t_2 + 3 \cdot d$ all correct nodes have invoked pulses, and hence have $WantToPulse$ as 0 for at least $\text{CYCLE}_{main} - 3 \cdot d$ time units. Therefore, in the interval $[t_2 + 3 \cdot d, t_2 + \text{CYCLE}_{main} - 3 \cdot d]$ no correct node joins any instance of$\mathcal{Q}$ with an input value of 1. Since $\text{CYCLE}_{main} \geq \Delta_{max} + 9 \cdot d$, it holds that $t_2 + 3 \cdot d \in [t_e, t_s + \text{CYCLE}_{main}]$, hence during the time interval $[t_e, t_2 + \text{CYCLE}_{main} - 3 \cdot d]$ no correct node joins any $\mathcal{Q}$ instance with an input value of 1. Hence, in the time interval $[t_e + \Delta_{max}, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$ no correct node invokes a pulse. Since $t_1 + 3 \cdot d + \Delta_{max} \geq t_e + \Delta_{max}$ and since t_2 is the last time p invoked a pulse before $t_1 + 3 \cdot d + \Delta_{max}$, it holds that p did not invoke a pulse in the time-interval $[t_2, t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}]$.

Lastly, after $t_2 + 3 \cdot d$ time units have elapsed, all correct nodes have reset TIMER_{main} and TIMER_{start}. Since $\text{CYCLE}_{main} = \text{CYCLE}_{start}$, we have that when TIMER_{start} elapses at some correct node, then $WantToPulse = 1$ at that correct node. By time $t_2 + 3 \cdot d + \text{CYCLE}_{main}$ all correct nodes have set their value of $WantToPulse$ to 1. Consider the last correct node to do so, it starts executing $\mathcal{Q}$, as instructed by Line 1 (the elapsing of TIMER_{start}), and since the input values of all correct nodes are 1, it terminates with an output value of 1. This happens no later that $t_2 + 6 \cdot d + \text{CYCLE}_{main} + \Delta_{max}$. That is, p invokes a pulse no later than $t_3 = t_2 + 6 \cdot d + \text{CYCLE}_{main} + \Delta_{max}$. □

Note that the above lemma shows that a correct node p invokes a pulse in some pattern. That is, after each pulse there is a period of uncertainty, and afterwards there is a long period of no PULSEing. Then p invokes a pulse again, and so on. Note that in the above lemma, the TIMER_{end} was never used; it will be used in the following lemma, which claims the "uncertainty" period is of a

constant length, and at the end of it a pulse is invoked. This lemma will give us the required properties, since with it the PULSEing pattern of a correct node p will be constant, and since the system pulses-in-pattern, the entire PULSEing pattern of the system will be determined.

Lemma 7. *Consider t_1, t_2 to be as defined in Lemma 6. Once the system pulses-in-unison, the value of t_2 is in the interval $[t_1 + \Delta_{min} - 13 \cdot d, t_1 + \Delta_{max} + 3 \cdot d]$.*

Proof. According to Lemma 6, after the last pulse there is a silent period during which TIMER_{main} and TIMER_{start} tick away. Once they elapse (they both elapse together), the following happens. First, $WantToPulse$ is set to 1 (until the next pulse). Second, TIMER_{end} is reset; and third, a $\mathcal{Q}$ instance is initiated.

Since the system pulses-in-unison, after the last pulse (at time t_2) all correct nodes reset TIMER_{start}. This means that TIMER_{start} elapses at all correct nodes within a $3 \cdot d$ interval, which implies that TIMER_{end} elapses at all correct nodes within a $3 \cdot d$ interval. Consider the last node q to have had TIMER_{start} elapse (at time t'). No correct node has had TIMER_{start} elapse before time $t' - 3 \cdot d$, hence at time $t' - 3 \cdot d + \Delta_{min}$ all correct nodes still have $WantToPulse = 1$ (no $\mathcal{Q}$ instance managed to finish yet). Therefore, when q's TIMER_{end} elapses at time $t' - 10 \cdot d + \Delta_{min}$ (since $\text{CYCLE}_{end} = \Delta_{min} - 10 \cdot d$), all correct nodes are guaranteed to join q's $\mathcal{Q}(q)$ instance with input value of 1, and hence in time interval $[t' + \Delta_{min} - 10 \cdot d + \Delta_{min}, t' + \Delta_{min} - 7 \cdot d + \Delta_{max}]$ q invokes a pulse.

t'_1, t'_2 represent the same meaning as t_1, t_2, just for the "PULSEing cycle" that starts after t_2. Consider t'_1 to be the first time value at which a correct q' invokes a pulse after t_2 (note that according to Lemma 6, $t'_1 \in [t_2 + \text{CYCLE}_{main} - 3 \cdot d + \Delta_{min}, t_3]$). For q' to invoke a pulse, at least one correct node should have $WantToPulse = 1$ in the interval $[t'_1 - \Delta_{max}, t'_1 - \Delta_{min}]$. Since t'_1 is the first time some node invokes a pulse, and since $t' - 3 \cdot d$ is the first time some correct node has $WantToPulse = 1$ in the current "PULSEing cycle", we have that $t' \in [t'_1 - \Delta_{max}, t'_1 - \Delta_{min} + 3 \cdot d]$. Therefore, q invokes a pulse due to TIMER_{end}'s elapsing is in the interval $[t'_1 - \Delta_{max} + \Delta_{min} - 10 \cdot d + \Delta_{min}, t'_1 - 4 \cdot d + \Delta_{max}]$. Since $\Delta_{max} = \Delta_{min} + 3 \cdot d$, we have that the above time interval is $[t'_1 - 13 \cdot d + \Delta_{min}, t'_1 - 4 \cdot d + \Delta_{max}]$. This implies that $t'_2 \geq t'_1 - 13 \cdot d + \Delta_{min}$. Which means that starting from the first pulse, the next "PULSEing cycle" will have that $t_2 \geq t_1 - 13 \cdot d + \Delta_{min}$. □

The above lemmata show that if the system has been coherent for $\Delta_{\text{ERRATIC-PULSER}} := \Delta_{\mathcal{Q}} + 3 \cdot Cycle$, then all correct nodes invoke pulses together, and they have a distinctive PULSEing pattern: say a node invokes a pulse at some time t_1; during the interval $[t_1, t_1 + \Delta_{min} - 13 \cdot d]$ there could be some additional pulses, then during the interval $[t_1 + \Delta_{min} - 13 \cdot d, t_1 + \Delta_{max} + 3 \cdot d]$ at least one pulse is invoked, and then there is an interval of at least $\text{CYCLE}_{main} + \Delta_{min} - 3 \cdot d$ time units during which no pulse is invoked, and finally, within the next $12 \cdot d$ there will be new pulses and a new PULSEing "cycle" will start.

Note that the length of this "cycle" is bounded from below by $\Delta_{max} + \text{CYCLE}_{main} + \Delta_{min}$, and bounded from above by $\Delta_{max} + \text{CYCLE}_{main} + \Delta_{min} + 12 \cdot d$. In addition, notice that each such "cycle" starts with a "possibly noisy period"

of length $\Delta_{max}+3\cdot d$, and ends with a "quiet period" of $\text{CYCLE}_{main}+\Delta_{min}-3\cdot d$ time. Since $\text{CYCLE}_{main} \geq \Delta_{max}+9\cdot d$, we have that the quiet period is at least Δ_{min} longer than the first period. This remark is important for the next section.

8 Creating the Balanced-Pulser

The above Erratic-Pulser synchronizes the correct nodes into some repetitive pulseing pattern. However, to solve the pulse-synchronization problem, an additional algorithm is required. We now present the Balanced-Pulser algorithm, which starting from an arbitrary state, shortly after the system is coherent, produces pulses approximately once in a *Cycle*, despite the permanent presence of *Byzantine* nodes.

Algorithm **Balanced-Pulser** */* executed at node p */*

1. **execute** an instance $\mathcal{A}$ of Erratic-Pulser in the background;
2. **when** $\mathcal{A}$ produces a pulse
   if $\mathcal{A}$ has not produced a pulse for at least $\text{CYCLE}_{main}+\Delta_{min}-3\cdot d$ time, invoke a pulse.

Fig. 3. An algorithm solving the pulse-synchronization problem

Theorem 1. *Algorithm* Balanced-Pulser *solves the* pulse*-synchronization problem in a self-stabilizing and* Byzantine *tolerant manner.*

Proof. Once the system is coherent for $\Delta_{\mathcal{Q}}$ time, by Lemma 4 the system pulses-in-unison. Hence, each time a correct node sees $\mathcal{A}$ pulseing, within $3\cdot d$ time units all other correct nodes see the same. In addition, by Lemma 6 and Lemma 7, the pulses that $\mathcal{A}$ produces have a distinct pattern. That is, a pulse, then a period of length $\Delta_{max}+3\cdot d$ with possible pulses and a period of length $\text{CYCLE}_{main}+\Delta_{min}-3\cdot d$ with no pulses. Then, within $12\cdot d$, another pulse.

If a correct node hasn't heard $\mathcal{A}$ producing a pulse for $\text{CYCLE}_{main}+\Delta_{min}-3\cdot d$ time, it must mean that $\mathcal{A}$ has undergone the "quiet period", since the "possible noisy period" is short. Hence, the next pulse produced must be the beginning of a new "cycle". Therefore, all correct nodes invoke pulses together in Balanced-Pulser. In addition, all correct nodes invoke pulses only at the beginning of a "cycle", and they invoke pulses $3\cdot d$ apart of each other. Since all correct nodes invoke pulses only at the beginning of a "cycle", we need only to argue about the length of the "cycle".

According to the lemmata in the previous section, the difference between the "long-cycle" and the "short-cycle" is at most $12\cdot d$ time units. Setting $\text{CYCLE}_{min} := \Delta_{max}+\text{CYCLE}_{main}+\Delta_{min}$, and $\text{CYCLE}_{max} := \text{CYCLE}_{min}+12\cdot d$, we have that the system is in a synchronized_pulsing_state. That is, starting

from any state, the system reaches a synchronized_pulsing_state; this proves *convergence*. In addition, according to the previous section, the PULSEing pattern remains as long is the system is coherent, thus *closure* also holds. □

The convergence time of BALANCED-PULSER is the same as the convergence of ERRATIC-PULSER + *Cycle*; that is, $\Delta_{\mathcal{Q}} + 4 \cdot Cycle$ time units.

9 Discussion

Time complexity: Once the system has become coherent, the BALANCED-PULSER algorithm converges in $O(f) + O(Cycle)$ time.

Message complexity: The BALANCED-PULSER algorithm executes $2 \cdot (n - f)$ SS-BYZ-Q instances each *Cycle*. Since SS-BYZ-Q has $O(f \cdot n^2)$ message complexity, then the message complexity becomes $O(f \cdot n^3)$ per CYCLE.

Executing fewer ss-Byz-Q: The main feature of ERRATIC-PULSER is that "eventually there will be a correct node that executes SS-BYZ-Q". As presented, ERRATIC-PULSER has each correct node execute SS-BYZ-Q once its timers elapse. The algorithm can be adapted such that only $f + 1$ of the nodes (predetermined and considered as part of the program, not memory) can invoke $\mathcal{Q}$. Since there will always be a correct node that invokes $\mathcal{Q}$, the correctness of the algorithm holds. This reduces the message complexity to $O(f^2 \cdot n^2)$.

Clock synchronization: The Digital clock synchronization problem consists of having all correct nodes agree on an integer value that progresses linearly with time. To build a digital clock synchronization algorithm using a PULSEing algorithm, all that is needed is to execute an agreement on the next clock's value each time a pulse is invoked. Setting the cycle of the PULSE to be long enough for the agreement algorithm to terminate, ensures that all correct nodes will agree on the clock value, and advance it appropriately. Note that the convergence time of such an algorithm is the convergence time of the underlying PULSE algorithm, plus an additional $cycle_{max}$time units. See [4] for a more detailed discussion.

Arbitrary *Cycle* values: According to the constraints of the BALANCED-PULSER algorithm, *Cycle* must be larger than $2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$ time units. For the purpose of clock synchronization it is enough to have *Cycle* in the order of Δ; for example, $Cycle = 5 \cdot \Delta$ would suffice for a linear convergence of the digital clock synchronization algorithm.

However, if one wishes to use PULSEing for other reasons, it is desired to be able to PULSE in any *Cycle*. To PULSE every $Cycle' < 2 \cdot \Delta_{max} + \Delta_{min} + 9 \cdot d$, set *Cycle* to be some multiplication of $Cycle'$ such that it falls within the constraints. Now, each time that BALANCED-PULSER produces a pulse, reset a timer of $Cycle'$ long, and when it elapses, invoke a pulse and reset the timer again. The PULSEing pattern will be a pulse by BALANCED-PULSER every *Cycle* and $Cycle/Cycle'$ pulses in between. This scheme is similar to what is done in [18]. The tricky part is to notice that if a pulse is invoked less than $Cycle'$ time before a PULSE by BALANCED-PULSER then the timer for the "small" pulses is reset, and hence a pulse is invoked again only in $Cycle'$ time units. Note that the difference between $cycle_{max}$ and $cycle_{min}$ is still $12 \cdot d$, hence there is no meaning to having $Cycle' \leq 12 \cdot d$. That is, $Cycle'$ should always be larger than $12 \cdot d$ time units.

References

1. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
2. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
3. A. Daliot and D. Dolev. Self-stabilization of byzantine protocols. In *In Proc. of the 7th Symposium on Self-Stabilizing Systems (SSS'05)*, Barcelona, Spain, Oct 2005.
4. A. Daliot, D. Dolev, and H. Parnas. Linear time byzantine self-stabilizing clock synchronization. In *Proc. of 7th Int. Conference on Principles of Distributed Systems (OPODIS'03)*, La Martinique, France, Dec 2003. A corrected version appears in http://arxiv.org/abs/cs.DC/0608096.
5. Y. Sakurai, F. Ooshita, and T. Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *OPODIS*, pages 283–298, 2004.
6. M. Nesterenko and A. Arora. Tolerance to unbounded byzantine faults. In *SRDS*, pages 22–, 2002.
7. M. Nesterenko and A. Arora. Dining philosophers that tolerate malicious crashes. In *22nd Int. Conference on Distributed Computing Systems*, 2002.
8. S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004.
9. E. N. Hoch, D. Dolev, and A. Daliot. Self-stabilizing byzantine digital clock synchronization. In *Proc. of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, Dallas, Texas, Nov 2006.
10. A. Daliot and D. Dolev. Self-stabilizing byzantine agreement. In *In Proc. of the Twenty-fifth ACM Symposium on Principles of Distributed Computing (PODC'06)*, Denver, Colorado, Jul 2006.
11. B. Liskov. Practical use of synchronized clocks in distributed systems. In *Proceedings of 10th ACM Symposium on the Principles of Distributed Computing*, 1991.
12. A. Arora, S. Dolev, and M.G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.
13. S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Journal of Real-Time Systems*, 12(1):95–107, 1997.
14. S. Dolev and J. L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.
15. M. Papatriantafilou and P. Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.
16. T. Herman. Phase clocks for transient fault repair. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1048–1057, 2000.
17. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
18. D. Dolev and E. N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *In Proc. the 21st Int. Symposium on Distributed Computing (DISC'07)*, Lemesos, Cyprus, Sep. 2007.
19. M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.
20. A. Daliot and D. Dolev. Self-stabilizing byzantine pulse synchronization. Technical report, Cornell ArXiv, Aug 2005. url: http://arxiv.org/abs/cs.DC/0608092.
21. F. C. Freiling and S. Ghosh. Code stabilization. In *Proc. of the 7th Symposium on Self-Stabilizing Systems (SSS'05)*, Barcelona, Spain, Oct 2005.

A The use of SS-BYZ-AGREE

The mode of operation of the SS-BYZ-AGREE, a self-stabilizing Byzantine agreement protocol presented in [10] is as follows: A node that wishes to initiate agreement on a value does so by disseminating an initialization message to all nodes that will bring them to (explicitly) invoke the SS-BYZ-AGREE protocol. Nodes that did not invoke the protocol may join in and execute the protocol in case enough messages from other nodes are received during the protocol. The protocol requires correct initiating nodes not to disseminate initialization messages too often. In the context of the current paper, an (*Initiator*, p, *) message serves as the initialization message.

When the protocol terminates, the SS-BYZ-AGREE protocol returns (in each correct node q) a triplet (p, m, τ_q^p), where m is the agreed value that p has sent. The value τ_q^p is an estimate, on the receiving node q's local clock, as to when node p has sent its value m. We also denote it as the "recording time" of (p, m). Thus, a node q's decision value is $\langle p, m, \tau_q^p \rangle$ if the nodes agreed on (p, m). If the sending node p is faulty then some correct nodes may agree on $(p, \perp)$, where $\perp$ denotes a non-value, and others may not invoke the protocol at all. The function $rt(\tau_q)$ represents the time at which the local clock of q reads τ_q.

The SS-BYZ-AGREE protocol satisfies the following typical Byzantine agreement properties:

Agreement: If the protocol returns a value ($\neq\perp$) at a correct nodes, it returns the same value at all correct nodes;
Validity: If all correct nodes are triggered to invoke the protocol SS-BYZ-AGREE by a value sent by a correct node p, then all correct nodes return that value;
Termination: The protocol terminates within a finite time;

The proof uses the following properties of the SS-BYZ-AGREE protocol ([10]):

Timeliness-Agreement Properties:

1. (agreement) For every two correct nodes q and q' that decide $\langle p, m, \tau_q^p \rangle$ and $\langle p, m, \tau_{q'}^p \rangle$ at local times τ_q and $\tau_{q'}$ respectively: $|rt(\tau_q) - rt(\tau_{q'})| \leq 3d$.
2. (validity) If all correct nodes invoked the protocol in the interval $[t_0, t_0 + d]$, as a result of some initialization message containing m sent by a correct node p that spaced the sending by at least $6d$ from the completion of the last agreement on its message, then for every correct node q, the decision time τ_q, satisfies $t_0 - d \leq rt(\tau_q) \leq t_0 + 3d$.
3. (termination) The protocol terminates within Δ time units following its explicit invocation, and within $\Delta + 7d$ time units, in case it was not explicitly invoked [7].
4. (separation) Let q be any correct node that decided on any two agreements regarding p at local times τ_q and $\bar{\tau}_q$, then $t_2 + 5d < \bar{t}_1$ and $rt(\tau_q) + 5d < \bar{t}_1 < rt(\bar{\tau}_q)$, where t_2 is the latest time at which a correct node invoked SS-BYZ-AGREE in the earlier agreement, and $\bar{t}_1$ is the earliest time that SS-BYZ-AGREE was invoked by a correct node in the later agreement.

[7] $\Delta := 7(2f + 3)d$