Self-stabilizing Byzantine Digital Clock Synchronization

Ezra N. Hoch, Danny Dolev and Ariel Daliot
The Hebrew University of Jerusalem

We present a scheme that achieves self-stabilizing *Byzantine* digital clock synchronization assuming a "synchronous" system. This synchronous system is established by the assumption of a common external "beat" delivered with a regularity in the order of the network message delay, thus enabling the nodes to execute in lock-step. The system can be subjected to severe transient failures with a permanent presence of *Byzantine* nodes. Our algorithm guarantees eventually synchronized digital clock counters, i.e. common increasing integer counters associated with each beat. We then show how to achieve regular clock synchronization, progressing at real-time rate and with high granularity, from the synchronized digital clock counters.

There is one previous self-stabilizing *Byzantine* clock synchronization algorithm that also converges in linear time (relying on an underlying distributed pulse mechanism), but it requires to execute and terminate *Byzantine* agreement in between consecutive pulses. That algorithm, although it does not assume a synchronous system, cannot be easily transformed to benefit from the stronger synchronous system model in which the pulses (beats) are in the order of the message delay time apart. The only other previous self-stabilizing *Byzantine* digital clock synchronization algorithm operating in a similar synchronous model converges in expected exponential time. Our algorithm converges (deterministically) in linear time and utilizes the synchronous model to achieve optimal precision and a simpler algorithm. Furthermore, it does not require the use of local physical timers in order to synchronize the clock counters.

Categories and Subject Descriptors: D.1.3 [**Programming techniques**]: Concurrent Programming—*distributed programming*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Self Stabilization, Byzantine faults, Clock Synchronization, Digital Clock Synchronization, Pulse Synchronization

1. INTRODUCTION

Clock synchronization is a very fundamental task in distributed systems. The vast majority of distributed tasks require some sort of synchronization; and clock synchronization is a very straightforward and intuitive tool for supplying this. It thus makes sense to require an underlying clock synchronization mechanism to be

Author's address: School of Engineering and Computer Science, The Hebrew University of Jerusalem, 91904, Israel. {ezraho,dolev,adaliot}@cs.huji.ac.il
Part of the work was done while the second author visited Cornell University. This research was supported in part by ISF, ISOC, NSF, CCR, and AFSOR.

highly fault-tolerant. A self-stabilizing algorithm seeks to attain synchronization once lost; a *Byzantine* algorithm assumes synchronization is never lost and focuses on containing the influence of the permanent presence of faulty nodes.

We consider a system in which the nodes execute in lock-step by regularly receiving a common "pulse" or "tick" or "beat". We will use the "beat" notation in order to stay clear of any confusion with "pulse synchronization" or "clock ticks". Should the beat interval be at least as long as the bound on the execution-time for terminating *Byzantine* agreement, then the system becomes, in a sense, similar to classic non-stabilizing systems, in which algorithms are initialized synchronously.[1] On the other hand, should the pulse interval length be in the order of the communication end-to-end delay, then the problem becomes agreeing on beat-counters or on "special" beats among the frequent common beats received.

The digital clock synchronization problem is to ensure that eventually all the correct nodes hold the same value of the beat counter (*digital clock*) and as long as enough nodes remain correct, they will continue to hold the same value and to increase it by "one" following each beat.

The mode of operation of the scheme proposed in this paper is to initialize at every beat *Byzantine* consensus on the digital clock value. Thus, after a number of beats (or *rounds*), which equals the bound for terminating *Byzantine* consensus, say Δ rounds, all correct nodes have identical views on an agreed digital clock value of Δ rounds ago. Based on this global state, a decision is taken at every node how to adjust its local digital clock. At each beat, if the two most recently terminated consensus instances show consecutive digital clock values, then the node increments its digital clock and initializes a new consensus instance on this updated digital clock value. If the digital clocks are not synchronized, the new consensus instance is initialized with the "zero" value. The algorithm converges within $3 \cdot \Delta$ rounds. In this paper, the terms "clock" and "digital clock" are used interchangeably.

Related work: The presented self-stabilizing *Byzantine* clock synchronization algorithm assumes that common beats are received synchronously (simultaneously) and in the order of the message delay apart. The clocks progress at real-time rate. Thus, when the clocks are synchronized, in-spite of permanent *Byzantine* faults, the clocks may accurately estimate real-time.[2] Following transient failures, and with on-going *Byzantine* faults, the clocks will synchronize within a finite time and will progress at real-time rate, although the actual clock-reading values might not be directly correlated to real-time. Many applications utilizing the synchronization of clocks do not really require the exact real-time notion (see [Liskov 1991]). 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. Clock synchronization in a similar model has earlier been denoted as "digital clock synchronization" ([Arora et al. 1991; Dolev 1997; Dolev and Welch 1997; Papatriantafilou and Tsigas 1997]) or "synchronization of phase-clocks" ([Herman 2000]), in which the goal is to agree on continuously incrementing counters associated with the beats. The convergence

[1] See [Daliot et al. 2003a] for a self-stabilizing *Byzantine* clock synchronization algorithm, which executes on top of a distributed self-stabilizing *Byzantine* pulse-synchronization primitive.

[2] All the arguments apply also to the case where there is a small bounded drift among correct clocks.

time in those papers is not linear, whereas in our solution it is linear.

The additional requirement of tolerating permanent *Byzantine* faults poses a special challenge for designing self-stabilizing distributed algorithms due to the capability of malicious nodes to hamper stabilization. This difficulty may be indicated by the remarkably few algorithms resilient to both fault models (see [Daliot and Dolev 2005a] for a short review). The digital clock synchronization algorithms in [Dolev and Welch 2004] are, to the best of our knowledge, the first self-stabilizing algorithms that are tolerant to *Byzantine* faults. The randomized algorithm, presented in [Dolev and Welch 2004], operating in the same model as in the current paper, converges in expected exponential time.

In [Daliot et al. 2003a] we have previously presented a self-stabilizing *Byzantine* clock synchronization algorithm, which converges in linear time and does not assume a synchronous system. That algorithm executes on top of an internal pulse synchronization primitive with intervals that allow to execute *Byzantine* agreement in between. The solution presented in the current paper also converges in linear time and only assumes that the (external synchronously received) beats are on the order of the message delay apart. In [Daliot et al. 2003b] and [Daliot and Dolev 2005b] two pulse synchronization procedures are presented that do not assume any sort of prior synchronization such as common beats. The former is biologically inspired and the latter utilizes a self-stabilizing Byzantine agreement algorithm developed in [Daliot and Dolev 2006]. Both of these pulse synchronization algorithms are complicated and have complicated proofs, while the current solution is achieved in a relatively straightforward manner and its proofs are simpler. Due to the relative simplicity of the algorithm, formal verification methods, as were used in [Malekpour and Siminiceanu 2006], can potentially be used to formally verify the correctness of the proposed algorithm. An additional advantage of the current solution is that it can be implemented without the use of local physical timers at the nodes. Local timers are only needed to achieve a high granularity of the synchronized clocks.

2. MODEL

We consider a fully connected network of n nodes. All the nodes are assumed to have access to a "global beat system" that provides "beats" with regular intervals. The communication network and all the nodes may be subject to severe transient failures, which might eventually leave the system in an arbitrary state. The algorithm tolerates a permanent fraction, $f < \frac{n}{4}$, of faulty *Byzantine* nodes.

We say that a node is *Byzantine* if it does not follow the instructed algorithm and non-*Byzantine* otherwise. Thus, a node that has crashed or experiences some other fault that does not allow it to exactly follow the algorithm as instructed, is considered *Byzantine*, even if it does not behave maliciously. A non-*Byzantine* node will therefore be called *non-faulty*.

We assume that the network has bounded time on message delivery when it behaves coherently. Nodes are instructed to send their messages immediately after the delivery of a beat from the global beat system. We assume that message delivery and the processing involved can be completed between two consecutive global beats. More specifically, the time required for message delivery and message processing is called a *round*, and we assume that the time interval between global beats is greater

than and in the order of such a round.

At times of transient failures there can be any number of concurrent *Byzantine* faulty nodes; the turnover rate between faulty and non-faulty behavior of the nodes can be arbitrarily large and the communication network may behave arbitrarily. Eventually the system behaves coherently again; in such a situation the system may find itself in an arbitrary state.

Definition 2.1. The system is *coherent* if there are at most f Byzantine nodes, messages sent immediately following a beat, arrive and are processed at their non-faulty destinations before the next beat.

Since a non-faulty node may find itself in an arbitrary state, there should be some time of continuous non-faulty operation before it can be considered "correct".

Definition 2.2. A non-faulty node is considered *correct* only if it remains non-faulty for Δ_{node} rounds during which the system is coherent.[3]

Denote by $DigiClock_p(r)$ the value of the digital clock at node p at beat r. We say that the system is in a *synchronized_state* if for all correct nodes the value of their $DigiClock$ is identical.

Definition 2.3. **The digital-clock synchronization problem**
Convergence: Starting from an arbitrary system state, the system reaches a synchronized_state after a finite time.

Closure: If at beat r the system is in a synchronized_state then for every r', $r' \geq r$,

(1) the system is in a synchronized_state at beat r'; and
(2) $DigiClock(r') = (DigiClock(r) + r' - r) \pmod{max\text{-}clock}$,[4] at each correct node.

Note that the algorithm parameters n, f, as well as the node's id are fixed constants and thus considered part of the incorruptible correct code. Thus we assume that non-faulty nodes do not hold arbitrary values of these constants.

Remark 2.4. The above problem statement implicitly requires that the precision of the clock synchronization is zero. That is, an algorithm that solves the above problem would have all correct nodes agree on the same value of $DigiClock$ at the same time.

Previous works (such as [Daliot et al. 2003a]) that operate in a different model (without a distributed external "beat" system) would not achieve precision zero, even if executed in the current model. Hence, the proposed algorithm herein fully utilized the strength of the "global beat system" model.

2.1 The Byzantine Consensus Protocol

Our digital clock synchronization algorithm utilizes a *Byzantine* consensus protocol as a sub-routine. We will denote this protocol by $\mathcal{BC}$. We require the regular three

[3]The assumed value of Δ_{node} in the current paper will be defined later.
[4]"*max-clock*" is the wrap around of the variable *DigiClock*. All increments to *DigiClock* in the rest of the paper are assumed to be $\pmod{max\text{-}clock}$.

conditions of Consensus from $\mathcal{BC}$, and one additional fourth requirement. That is, in $\mathcal{BC}$ the following holds:

(1) Agreement: All non-faulty nodes terminate $\mathcal{BC}$ with the same output value.
(2) Validity: If all non-faulty nodes have the same initial value v, then the output value of all non-faulty nodes is v.
(3) Termination: All non-faulty nodes terminate $\mathcal{BC}$ within Δ rounds.
(4) Solidarity. If the non-faulty nodes agree on a value v, such that $v \neq \perp$ (where $\perp$ denotes a non-value), then there are at least $n - 2 \cdot f$ non-faulty nodes with initial value v.

Remark 2.5. Note that for $n > 4f$ the "solidarity" requirement implies that if the *Byzantine* consensus is started with at most $\frac{n}{2}$ non-faulty nodes with the same value, then all non-faulty nodes terminate with the value $\perp$.

As we commented above, since $\mathcal{BC}$ requires the nodes to maintain a consistent state throughout the protocol, a non-faulty node that has recently recovered from a transient fault cannot be considered correct. In the context of this paper, a non-faulty node is considered *correct* once it remains non-faulty for at least $\Delta_{node} = \Delta + 1$ beats and as long as it continues to be non-faulty.

In Section 3 we discuss how a typical synchronous *Byzantine* consensus protocol can be used as such a $\mathcal{BC}$ protocol. The specific example we discuss has two early stopping features: First, termination is achieved within $2f + 4$ of our rounds. If the number of actual *Byzantine* nodes is $f' \leq f$ then termination is within $2f' + 6$ rounds. Second, if all non-faulty nodes have the same initial value, then termination is within 4 rounds.

The symbol Δ denotes the bound on the number of rounds it takes $\mathcal{BC}$ to terminate at all correct nodes. That is, if $\mathcal{BC}$ has some early stopping feature, we still wait until Δ rounds pass. This means that the early stopping may improve the message complexity, but not the time complexity. By using the protocols in Section 3, we can set $\Delta := 2f + 4$ rounds.

3. *BYZANTINE* CONSENSUS ($\mathcal{BC}$) IMPLEMENTATION

We present a $\mathcal{BC}$ protocol, denoted Byz-Consensus, that has the four properties required by our algorithm. This is done by converting the *Byzantine* agreement protocol in [Toueg et al. 1987] into a *Byzantine* consensus protocol. In the following sections we show that the converted protocol guarantees *agreement, termination, validity* and *solidarity*.

We start by presenting our $\mathcal{BC}$ protocol in Figure 1. In the following subsection we claim that the *broadcast* primitive continues to hold its properties in the new protocol. The full proofs are in Appendix A.

3.1 The $\mathcal{BC}$ Protocol

The difference between a *Byzantine* agreement protocol and a *Byzantine* consensus protocol is that in agreement there is a *general* G with some initial value v, and all correct nodes need to agree on G's value. In consensus, there is no general, and every correct node has its own initial value, and all correct nodes need to agree on an output value. *Byzantine* consensus "can be seen" as an agreement problem

where the *general* might have sent different values to different nodes, and all correct nodes need to agree on what value G has sent.

We use the *Byzantine* agreement protocol in [Toueg et al. 1987], and alter it in the following way. We consider the general to be a virtual node I_0, that sends its value to all nodes in the first round. I_0 does not participate in the protocol, except for "supposedly" having "sent" its value in the first round. Other than this change, our protocol is almost identical to the protocol in [Toueg et al. 1987]. The added change transforms the protocol in [Toueg et al. 1987] to a *Byzantine* consensus protocol, and also ensures that *solidarity* holds. Note that I_0 is not considered a *Byzantine* node or a correct node, it has a special status of a "virtual" node. That is, neither n (the number of nodes in the system) nor f count I_0.

We achieve the *solidarity* requirement by sending the initial value of every node to all other nodes. We say that two messages "(v)" are *distinct* if they were sent by different nodes. An *echo* message is sent only if a correct node received $n - f$ distinct "(v)" messages. Hence, there were at least $n - 2 \cdot f$ nodes with v as their initial value. Clearly, no correct node will accept if not even a single correct node has sent an *echo* message. Therefore, if a correct node accepted some value v', then an *echo* message was sent by a correct node, and that means that at least $n - 2 \cdot f$ correct nodes had v' as their initial value.

Protocol **Byz-Consensus** */* executed at node p where m is the initial value */*

(1) $broadcasters := \phi; v := \perp$;
(2) phase = 1:
  send(m) to all nodes;
(3) phase = 2:
 (a) **if received** $n - f$ distinct (v') messages by end of phase 1 **then**
  send(echo, I_0, v', 1) to all nodes;
 (b) **if received** $n - f$ distinct (echo, I_0, v', 1) messages by the end of phase 2 **then**
  $v = v'$;
(4) **round** r **for** $r = 2$ **to** $r = f + 2$ **do**:
 (a) **if** $v \neq \perp$ **then**
  invoke **broadcast**(p, v, r);
  stop and **return**(v);
 (b) by the end of round r:
  if in round $r' \leq r$ **accepted**$(I_0, v', 1)$ and (q_i, v', i) for all $i, 2 \leq i \leq r$,
   where all q_i distinct **then**
  $v := v'$;
 (c) **if** $|broadcasters| < r - 1$ **then**
  stop and **return**(v);
(5) **stop** and **return**(v);

Fig. 1. The Byzantine Consensus algorithm

The protocol we present here is round based, and each round consists of two phases. In Appendix A, Subsection A.2, we prove the following: *The "Agreement", "Validity", "Termination" and "Solidarity" conditions hold for Byz-Consensus.*

3.2 The *b*roadcast Primitive

We use the same *broadcast* primitive as in [Toueg et al. 1987]. The *broadcast* primitive is provided here for convenience. It is almost an exact copy of the *broadcast* primitive in [Toueg et al. 1987], with minor changes for readability.

Broadcast Primitive /* *rules for broadcasting and accepting* (p, m, k) */

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 $(echo, p, m, k)$ 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 in phase $2k$
    then send $(init', p, m, k)$ to all;
    if received $(init', p, m, k)$ from $\geq n - 2f$ distinct nodes in phase $2k+1$
    then $broadcastuers := brodcasters \cup \{p\}$;
  Phase $2k+2$:
    if received $(init', p, m, k)$ from $\geq n - f$ distinct nodes in phase $2k+1$
    then send $(echo', p, m, k)$ to all;
    if received $(echo', p, m, k)$ from $\geq n - f$ distinct nodes in phase $2k+2$
    then *accept* (p, m, k);
Round $r \geq k+2$:
  Phase $2r-1, 2r$:
    if received $(echo', p, m, k)$ from $\geq n - 2f$ distinct nodes in previous phases,
      and not sent $(echo', p, m, k)$
    then send $(echo', p, m, k)$ to all;
    if received $(echo', p, m, k)$ from $\geq n - f$ distinct nodes in previous phases
    then *accept* (p, m, k);

Fig. 2. The *broadcast* primitive from

A minor difference needs to be addressed in the *broadcast* primitive. Instead of allowing the acceptance of messages only from the set of nodes P, we also allow accepting a message from a single virtual node, named I_0. That is, a node may accept a message $(I_0, v, 1)$. However, since I_0 is not an actual node, it does not participate in sending messages or receiving them.

When stating (and proving) the properties of the *broadcast* primitive, we consider $I_0 \notin P$. We consider $n = |P|$ and f to be the number of *Byzantine* nodes **not** including I_0.

Note that an *init* message for I_0 is never sent, because I_0 is a virtual node, and its *init* message is simulated by the first phase of the Byz-Consensus protocol.

Also note that the messages sent in line 3.a and received in line 3.b are of the same format as messages sent by the *broadcast* primitive. That is, nodes may accept a message $(I_0, v, 1)$, "as if" I_0 actually sent it. Hence, we cannot simply use the

original *broadcast* primitive as a "black box", even though it has not been changed. We are required to reprove the properties shown in [Toueg et al. 1987].

The properties that hold regarding the *broadcast* primitive used in our protocol, are:

(1) *Correctness*: If a correct node $p \in P$ executes broadcast(p, m, k) in round k, then every correct node accepts (p, m, k) in the same round.

(2) *Unforgeability*: If a correct node $p \in P$ does not execute broadcast(p, m, k), then no correct node ever accepts (p, m, k).

(3) *Relay*: If a correct node accepts (p, m, k) in round r for some $p \in P \cup \{I_0\}$, 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, for some $p \in P \cup \{I_0\}$, then every correct node has $p \in broadcasters$ at the end of round $k + 1$.

The proofs of the above properties are in Appendix A, Subsection A.1.

4. DIGITAL CLOCK SYNCHRONIZATION ALGORITHM

The following digital clock synchronization algorithm tolerates up to $f < \frac{n}{4}$ concurrent *Byzantine* faults. The objective is to have the digital clocks increment by "1" every beat and to achieve synchronization of these digital clocks.

4.1 Intuition for the Algorithm

The idea behind our algorithm is that each node runs many simultaneous *Byzantine* consensus protocols. In each round of the algorithm it executes a single round in each of the *Byzantine* consensus protocols, but each *Byzantine* consensus protocol instance is executed with a different round number. That is, if $\mathcal{BC}$ takes Δ rounds to terminate, then each node runs Δ concurrent instances of it, where, for the first one it executes the first round, for the second it executes the second round, and in general for the i^{th} $\mathcal{BC}$ protocol it executes the i^{th} round. We index a $\mathcal{BC}$ protocol by the number of rounds passed from its invocation. When the Δ^{th} $\mathcal{BC}$ protocol is completed, a new instance of $\mathcal{BC}$ protocol is initiated. This mechanism, of executing concurrently Δ $\mathcal{BC}$ protocols, allows the non-faulty nodes to agree on the clock values as of Δ rounds ago. The nodes use the consistency of these values as of Δ rounds ago together with the exchange of their current values to "tune" the future clock values.

4.2 Preliminaries

Given a *Byzantine* consensus protocol $\mathcal{BC}$, each node maintains the following variables and data structures:

(1) *DigiClock* holds the beat counter value at the node.

(2) *ClockVec* holds a vector containing the value of *DigiClock* that each node sent in the current round.

(3) $DigiClock_{most}$ holds the value that appears at least $\frac{n}{2} + 1$ times in *ClockVec,* if one exists.

(4) $Agree[i]$ is the memory space of the i^{th} instance of $\mathcal{BC}$ protocol (the one initialized i rounds ago).

(5) v holds the agreed value of the currently terminating $\mathcal{BC}$.

(6) v_{prev} holds the value of v one round ago.

Note that all the variables are reset or recomputed periodically, so even if a node begins with arbitrary values in its variables, it will acquire consistent values. Similarly, the memory space of $\mathcal{BC}$ is reset whenever a node initiates and starts to execute a new $\mathcal{BC}$ instance. The consistency of the variable values used for $\mathcal{BC}$ are taken care of within that protocol.

Figure 3 presents the digital clock synchronization algorithm.

Algorithm **Digital-SSByz-ClockSync** */* executed at each beat */*

(1) for each $i \in \{1, .., \Delta\}$ do
execute the i^{th} round of the $Agree[i]$ BC protocol;

(2) send value of $DigiClock$ to all nodes and store the received clocks of other nodes in $ClockVec$;

(3) set the following:
  (a) v:= the agreed value of $Agree[\Delta]$;
  (b) $DigiClock_{most}$:= the value appearing at least $\lfloor \frac{n}{2} \rfloor + 1$ times in $ClockVec$, and 0 otherwise;

(4) (a) if $(v = 0)$ or $(v = v_{prev} + 1)$ then
  $DigiClock := DigiClock_{most} + 1 \pmod{max\text{-}clock}$;
  (b) else
  $DigiClock := 0$;

(5) for each $i \in \{2, ..., \Delta\}$ do
$Agree[i] := Agree[i-1]$;

(6) initialize $Agree[1]$ by invoking $\mathcal{BC}(DigiClock)$.

(7) $v_{prev} := v$.

Fig. 3. The digital clock synchronization algorithm

Remark 4.1. The model allows for only one message to be sent from node p to p' within one round (between two consecutive beats). The digital clock synchronization algorithm in Figure 3 requires sending two sets of messages in each round. Observe that the set of messages sent in Step 2 is not dependent on the operations taking place in Step 1, therefore, all messages sent by the algorithm during each round can be sent right after the beat and will arrive and processed before the next beat, meeting the model's assumptions.

Note that a "simpler" solution, such as running consensus on previous $DigiClock$, incrementing it by $\Delta+1$ and setting that as the current $DigiClock$ would not work, because for some specific initial values of $DigiClock$ the *Byzantine* nodes can cause the non-faulty nodes to get "stuck" in an infinite loop of alternating values.

5. LEMMATA AND PROOFS

All the lemmata, theorems, corollaries and definitions hold only as long as the system is coherent. We assume that all nodes may start in an arbitrary state, and that from some time on, at least $n - f$ of them are concurrently correct. We will denote by $\mathcal{G}$ a group of nodes that behave according to the algorithm, and that are not subject to (for some pre-specified number of rounds) any new transient faults. If, $|\mathcal{G}| \geq n - f$ and remain non-faulty for a sufficiently long period of time ($\Omega(\Delta)$ global beats), then the system will converge.

For simplifying the notations, the proof refers to some "external" round number. The nodes do not maintain it, it is only used for the proofs.

Definition 5.1. We say that the system is $\textsc{Calm}(\alpha, \sigma)$, $\sigma > \alpha$, if there is a set $\mathcal{G}$, $|\mathcal{G}| = n - f$, of nodes that are continuously non-faulty during all rounds in the interval $[\alpha, \sigma - 1]$.

The notation $\textsc{Calm}(\alpha, \sigma \geq \beta)$ denotes that $\textsc{Calm}(\alpha, \sigma)$ and $\sigma \geq \beta$. Specifically, the notation implies that the system was calm for at least β rounds. Notice that all nodes in $\mathcal{G}$ are considered correct when the system is $\textsc{Calm}(\alpha, \sigma \geq \Delta)$.

Note that in typical self-stabilizing algorithms it is assumed that eventually all nodes behave correctly, and therefore there is no need to define $\textsc{Calm}()$. In our context, since some nodes may never behave correctly, and additionally some nodes may recover and some may fail we need a sufficiently large subset of the nodes to behave correctly for sufficiently long time in order for the system to converge.

In the following lemmata, $\mathcal{G}$ refers to the set implied by $\textsc{Calm}(\alpha, \sigma)$, without stating so specifically.

Lemma 5.2. *If the system is* $\textsc{Calm}(\alpha, \sigma \geq \Delta + 1)$*, then for any round* β, $\beta \in [\alpha + \Delta + 1, \sigma]$, *all nodes in* $\mathcal{G}$ *have identical values of* v *after executing Step 2 of Digital-SSByz-ClockSync.*

Proof. Irrespective of the initial states of the nodes in $\mathcal{G}$ at the beginning of round α (which is after the last transient fault in $\mathcal{G}$ occurred), the beats received from the global beat system will cause all nodes in $\mathcal{G}$ to perform the steps in synchrony. By the end of round α, all nodes in $\mathcal{G}$ reset $\mathcal{BC}$ protocol *Agree*[1].

Note that at each round another $\mathcal{BC}$ protocol will be initialized and after Δ rounds from its initialization each such protocol returns the same value at all nodes in $\mathcal{G}$, since all of them are non-faulty and follow the protocol. Hence, After $\Delta + 1$ rounds, the values all nodes in $\mathcal{G}$ receive as outputs of $\mathcal{BC}$ protocols are identical. Therefore v is identical at all $g \in \mathcal{G}$, after executing Step 2 of that round.

Since this claim depends only on the last $\Delta + 1$ rounds being "calm", the claim will continue to hold as long as such a $\mathcal{G}$ set of nodes not experiencing transient faults exists. Thus, this holds for any round β, $\alpha + \Delta \leq \beta \leq \sigma$. □

Lemma 5.3. *If the system is* $\textsc{Calm}(\alpha, \sigma \geq \Delta + 2)$*, then for any round* $\beta \in [\alpha + \Delta + 1, \sigma]$, *either all nodes in* $\mathcal{G}$ *perform Step 4.a, or all of them perform Step 4.b.*

Proof. By Lemma 5.2, after the completion of Step 2 of round $\alpha + \Delta + 1$ the value of v is the same at all nodes of $\mathcal{G}$, hence after an additional round the value of

v_{prev} is the same at all nodes of $\mathcal{G}$. Since the decision whether to perform Step 4.a or Step 4.b depends only on the values of v, and v_{prev}, all nodes in $\mathcal{G}$ perform the same line (either 4.a or 4.b). Moreover, because this claim depends on the last $\Delta+2$ rounds being "calm", the claim will continue to hold as long as no node in $\mathcal{G}$ is subject to a fault. □

Denote $\Delta_1 := \Delta + 2$. All the following lemmata will assume the system is CALM(α, β), for rounds $\beta \geq \Delta_1$. Therefore, in all the following lemmata, we will assume that in each round β, all nodes in $\mathcal{G}$ perform the same Step 4.x (according to Lemma 5.3).

LEMMA 5.4. *If the system is* CALM$(\alpha, \sigma \geq \Delta_1)$*, and if at the end of some* $\beta \geq \alpha + \Delta_1 - 1$*, all nodes in* $\mathcal{G}$ *have the same value of DigiClock, then at the end of any* β'*,* $\beta \leq \beta' \leq \sigma$*, they will have the same value of DigiClock.*

PROOF. Since we consider only $\beta \geq \alpha + \Delta_1 - 1$, by Lemma 5.3 all nodes in $\mathcal{G}$ perform the same step in Step 4. For round $\beta' = \beta + 1$, the value of *DigiClock* can be changed at Lines 4.a or 4.b. If it was changed at 4.b then all nodes in $\mathcal{G}$ have the value 0 for *DigiClock*. If it was changed by Step 4.a, then because we assume that at round β all nodes in $\mathcal{G}$ have the same *DigiClock* value, and because $|\mathcal{G}| = n - f \geq \lfloor \frac{n}{2} + 1 \rfloor$, the value of $DigiClock_{most}$ computed at round β' is the same for all nodes in $\mathcal{G}$, and therefore, executing Step 4.a will produce the same value for *DigiClock* in round β' for all nodes in $\mathcal{G}$.

By induction, for any $\beta \leq \beta' \leq \sigma$, all nodes in $\mathcal{G}$ continue to agree on the value of *DigiClock*. □

Denote $\Delta_2 := \Delta_1 + \Delta + 1$. All the following lemmata will assume the system is CALM(α, σ), for $\sigma \geq \Delta_2$.

LEMMA 5.5. *If the system is* CALM$(\alpha, \sigma \geq \Delta_2)$*, then at the end of any round* $\beta, \beta \in [\alpha + \Delta_2 - 1, \sigma]$*, the value of DigiClock at all nodes in* $\mathcal{G}$ *is the same.*

PROOF. Consider any round $\beta' \in [\alpha + \Delta_1 - 1, \alpha + \Delta_1 + \Delta - 1]$. If at the end of β' all nodes in $\mathcal{G}$ hold the same *DigiClock* value, then from Lemma 5.4 this condition holds for any β, $\beta \in [\alpha + \Delta_2 - 1, \sigma]$. Hence, we are left to consider the case where at the end of any such β' not all the nodes in $\mathcal{G}$ hold the same value of *DigiClock*. This implies that Step 4.b was not executed in any such round β'. Also, if Step 4.a was executed during any such round β', and there was some *DigiClock* value that was the same at more than $\frac{n}{2}$ nodes in $\mathcal{G}$, then after the execution of Step 4.a, all nodes would have had the same *DigiClock* value. Hence, we assume that for all β', only Step 4.a was executed, and that no more than $\frac{n}{2}$ from $\mathcal{G}$ had the same *DigiClock* value.

Consider round $\beta'' = \alpha + \Delta_1 + \Delta$. The above argument implies that at round $\beta'' - \Delta$, Step 4.a was executed, and there were no more than $\frac{n}{2}$ nodes in $\mathcal{G}$ with the same *DigiClock* value. Since $\frac{n}{2} < n - 2 \cdot f$, the "solidarity" requirement of $\mathcal{BC}$ implies that the value entered into v at round β'' is $\perp$. Hence, at round β'' Step 4.b would be executed.

Therefore, during one of the rounds $\beta \in [\alpha + \Delta_1 - 1, \alpha + \Delta_1 + \Delta]$, all the nodes in $\mathcal{G}$ have the same value of *DigiClock*, and from Lemma 5.4 this condition holds for all rounds, until σ. □

Remark 5.6. The requirement that $f < \frac{n}{4}$ stems from the proof above. That is because we require that $\frac{n}{2} < n - 2 \cdot f$ (to be able to use the "solidarity" requirement of $\mathcal{BC}$). We note that this is the only place that the requirement $f < \frac{n}{4}$ appears, and that it is a question for future research whether this can be improved to the known lower bound of $f < \frac{n}{3}$.

COROLLARY 5.7. *If the system is* CALM$(\alpha, \sigma \geq \Delta_2)$*, then for every round* β, $\beta \in [\alpha + \Delta_2 - 1, \sigma - 1]$*, one of the following conditions holds:*

(1) *The value of DigiClock at the end of round* $\beta + 1$ *is "0" at all nodes in* $\mathcal{G}$.
(2) *The value of DigiClock at the end of round* $\beta + 1$ *is identical at all nodes in* $\mathcal{G}$ *and it is the value of DigiClock at the end of round* β *plus "1".*

LEMMA 5.8. *If the system is* CALM$(\alpha, \sigma \geq \Delta_2 + \Delta)$*, then for every round* $\beta \in [\alpha + \Delta_2 + \Delta - 1, \sigma]$*, Step 4.b is not executed.*

PROOF. By Corollary 5.7, for all rounds β, $\beta \in [\alpha + \Delta_2 - 1, \sigma - 1]$ one of the two conditions of the *DigiClock* values holds. Due to the "validity" property of $\mathcal{BC}$, after Δ rounds, the value entered into v is the same *DigiClock* value that was at the nodes in $\mathcal{G}$, Δ rounds ago. Therefore, after Δ rounds, the above conditions hold on the value of v, v_{prev}. Hence, for any round β, $\beta \in [\alpha + \Delta_2 + \Delta - 1, \sigma]$ one of the conditions holds on v, v_{prev}. Since for both of these conditions, Step 4.a is executed, Step 4.b is never executed for such a round β. □

COROLLARY 5.9. *If the system is* CALM$(\alpha, \sigma \geq \Delta_2 + \Delta)$*, then for every round* β, $\beta \in [\alpha + \Delta_2 + \Delta - 1, \sigma]$*, it holds that all nodes in* $\mathcal{G}$ *agree on the value of DigiClock and increase it by "1" at the end of each round.*

Corollary 5.9 implies, in a sense, the convergence and closure properties of algorithm Digital-SSByz-ClockSync.

THEOREM 5.10. *From an arbitrary state, once the system stays coherent and there are* $n - f$ *nodes that are non-faulty for* $3\Delta + 3$ *rounds, Digital-SSByz-ClockSync ensures convergence to a synchronized_state. Moreover, as long as there are at least* $n - f$ *correct nodes at each round the closure property also holds.*

PROOF. The conditions of the theorem implies that the system satisfies the property CALM$(\alpha, \sigma \geq \Delta_2 + \Delta)$. Consider the system at the end of round $\Delta_2 + \Delta$ and denote by $\bar{\mathcal{G}}$ a set of $n - f$ correct nodes implied by CALM$(\alpha, \sigma \geq \Delta_2 + \Delta)$. Consider all the Δ instances of $\mathcal{BC}$ in their memory. Denote by $\mathcal{BC}_i$ the instance of $\mathcal{BC}$ initialized i $(0 \leq i \leq \Delta - 1)$ rounds ago. By Lemma 5.8, Step 4.b is not going to be executed (if the nodes in $\bar{\mathcal{G}}$ will continue to be non-faulty). Therefore, at the end of the current round,

(1) the set of inputs to each $\mathcal{BC}_i$ contained at least $\lfloor \frac{n}{2} \rfloor + 1$ identical values from non-faulty nodes, when it was initialized (denote that value I_i);
(2) for every i, $0 \leq i \leq \Delta - 1$, either $I_i = I_{i+1}$ or $I_i = 0$;
(3) I_0 is the value that at least $\lfloor \frac{n}{2} \rfloor + 1$ non-faulty hold in their DigiClockat the end of the current round.

The first property holds because otherwise, by the "solidarity" property of $\mathcal{BC}$, the agreement in that $\mathcal{BC}$ will be on $\perp$ and Step 4.b will be executed. The second

property holds because otherwise Step 4.b will be executed. The third property holds since this is the value they initialized the last $\mathcal{BC}$ with.

Observe, that each $\mathcal{BC}_i$ will terminate in $\Delta - i$ rounds with a consensus agreement on I_i, as long as there are $n-f$ non-faulty nodes that were non-faulty throughout its Δ rounds of execution. Thus, under such a condition, for that to happen some nodes from $\bar{\mathcal{G}}$ may fail and still the agreement will be reached. Therefore, Corollary 5.9 holds for each node that becomes correct, i.e., was non-faulty for Δ rounds, because it will compute the same values as all the other correct nodes.

By a simple induction we can prove that the three properties above will hold in any future round, as long as for each $\mathcal{BC}$ there are $n - f$ non-faulty nodes that executed it.

Thus, the three properties imply that the basic claim in Corollary 5.9 will continue to hold, which completes the proof of the convergence and closure properties of the system. □

Note that if the system is "synchronized" and the actual number of *Byzantine* faults f' is less than f, then Theorem 5.10 implies that any non-faulty node that is not in $\mathcal{G}$ (there are no more than $f - f'$ such nodes) synchronizes with the $DigiClock$ value of nodes in $\mathcal{G}$ after at most Δ global beats from its last transient fault.

6. COMPLEXITY ANALYSIS

The clock synchronization algorithm presented above converges in $3 \cdot \Delta + 3$ rounds. That is, it converges in $\Omega(f)$ rounds (since $\Delta = 2 \cdot f + 4$ for our $\mathcal{BC}$ of choice).

Once the system converges, and there are at least $|\mathcal{G}| = n - f$ correct nodes, the $\mathcal{BC}$ protocol will stop executing after 4 rounds for all nodes in $\mathcal{G}$ (due to the early stopping feature of $\mathcal{BC}$ we use). During each round of $\mathcal{BC}$, there are n^2 messages exchanged. Note that we execute Δ concurrent $\mathcal{BC}$ protocols; hence, over a period of Δ rounds, there are $\Delta \cdot 4 \cdot n^2$ messages sent. Therefore, the amortized message complexity per round is $O(n^2)$. Note that the early stopping of $\mathcal{BC}$ does not improve the convergence rate. It only improves the amortized message complexity.

7. DISCUSSION

7.1 A Scheme for "Rotating Consensuses"

Although the current work is presented as a digital clock synchronization algorithm, it actually surfaces a more general scheme for "rotating" *Byzantine* consensus instances, which allows all non-faulty nodes to have a global "snapshot" of the state that was several rounds ago. This mechanism is self-stabilizing and tolerates the permanent presence of *Byzantine* nodes. This mechanism ensures that all non-faulty nodes decide on their next step at the next round, based on the same information.

Our usage of consensus provides agreement on a global "snapshot" of some global state. By replacing each *Byzantine* consensus with n *Byzantine* agreements, this mechanism can provide a global "snapshot" of the states of all the nodes several rounds ago. That is, instead of agreeing on a single state for the entire system, we would agree on the local state of each node. Every time the agreement instances terminate the nodes may evaluate a predicate that can determine whether the past global state was legal. If the global state is of a non-stabilizing algorithm $\mathcal{A}$, the

nodes may then decide whether to reset $\mathcal{A}$ accordingly; this would producing a building block for a general stabilizing mechanism.

The next subsection specifies some additional results which can be achieved using this scheme.

7.2 Additional Results

The digital clock synchronization algorithm presented here can be quickly transformed into a token circulation protocol in which the token is held in turn by any node for any pre-determined number of rounds and in a pre-determined order. The pre-determined variables are part of the required incorruptible code. E.g. if the token should be passed every k beats, then node p_i, $i = 1 + \lfloor \frac{DigiClock}{k} \rfloor \pmod{n}$ holds the token during rounds $[k \cdot (i-1) + 1, k \cdot i]$. Similarly, it can also produce synchronized pulses which can then be used to produce the self-stabilizing counterpart of general *Byzantine* protocols by using the scheme in [Daliot and Dolev 2005a]. These pulses can be produced by setting *max-clock* to be the pulse cycle interval, and issuing a pulse each time $DigiClock = 0$.

7.3 Digital Clock vs. Clock Synchronization

In the described algorithm, the non-faulty nodes agree on a common integer value, which is regularly incremented by one. This integer value is considered "the synchronized (digital) clock value". Note that clock values estimating real-time or real-time rate can be achieved in two ways. The first one, is using the presented algorithm to create a new distributed pulse, with a large enough cycle, and using the algorithm presented in [Daliot et al. 2003a] to synchronize the clocks. The second, is to adjust the local clock of each node, according to the value of the common integer value, multiplied by the predetermined length of the beat interval.[5] This way, at each beat of the global beat system, the clocks of all the nodes are incremented at a rate estimating real-time.

8. FUTURE WORK

We consider four main points to be of interest for future research.

(1) Can the tolerance of the algorithm be improved to support $f < \frac{n}{3}$?

(2) Can the above mechanism be applied in a more general way, leading to a general stabilizer of *Byzantine* tolerant algorithms without using the scheme proposed in [Daliot and Dolev 2005a], which requires pulses that are sufficiently spaced apart?

(3) What happens if the global beats are received at intervals that are less than the message delay, i.e. common clock ticks. Is there an easy solution to achieve synchronized clocks? If yes, can it attain optimal precision like the current solution? If no, is the only option then to synchronize the clocks in a fashion similar to [Daliot and Dolev 2005b; Daliot et al. 2003b; 2003a]? i.e. by executing an underlying distributed pulse primitive with pulses that are far enough apart in order to be able to terminate agreement in between. In that case, is

[5]This value need also be defined as part of the incorruptible code of the nodes.

there any advantage in having a common source of the clock ticks or is it simply a replacement for the local timers of the nodes?

(4) The proposed algorithm requires that all nodes are connected to each other, or more specifically that the diameter of the network graph is 1 and that the minimal degree of a node is at least $2 \cdot f + 1$ (see [Dolev 1982] for a more detailed discussion about the $2 \cdot f + 1$ requirement). The requirement of "diameter 1" appears in all previous self-stabilizing *Byzantine* tolerant clock synchronization results. What happens if the diameter is not 1? Can clock synchronization be solved in such a scenario in a self-stabilizing and *Byzantine* tolerant manner?

A. APPENDIX: PROOFS FOR $\mathcal{BC}$ PROTOCOL

We now present the proof for the protocol Byz-Consensus presented in Section 3 along with the proofs for the correctness of the *broadcast* primitive. We start with the proof of the properties of the *broadcast* primitive; In the following subsection we prove the correctness of the Byz-Consensus algorithm.

A.1 Proof of the Properties of the *broadcast* Primitive

The proofs are very similar to the proofs given in [Toueg et al. 1987]. Where the proofs are identical, we refer the reader to the original proofs in [Toueg et al. 1987].

CLAIM A.1. *The* Correctness *property holds.*

PROOF. We consider *Broadcast* operations for some $p \in P$. Since $I_0 \notin P$, and I_0 does not send or receive any messages, the behavior of the *Broadcast* primitive regarding $p \in P$ is the same as in [Toueg et al. 1987]. Hence, the proof of the *Correctness* property holds, as in [Toueg et al. 1987]. □

CLAIM A.2. *The* Unforgeability *property holds.*

PROOF. The same considerations of the *Correctness* property hold here too and this property is proven the same as in [Toueg et al. 1987]. □

CLAIM A.3. *The* Relay *property holds.*

PROOF. For $p \in P$, the proof from [Toueg et al. 1987] holds. We consider $p = I_0$. Hence, we need to show that if a correct node p accepts $(I_0, m, 1)$ at round r, then every other correct node accepts $(I_0, m, 1)$ in round $r + 1$ or earlier.

If p accepted $(I_0, m, 1)$ in round 1, then p received more than $n - f$ $(echo, I_0, m, 1)$ messages. Hence, more than $n - 2 \cdot f$ were sent by correct nodes, which implies that all correct nodes received $n - 2 \cdot f$ such messages by the end of the second phase of round 1. Therefore, at the first phase of round 2, all correct nodes will send $(init', I_0, m, 1)$ to all nodes. Hence, at the second phase of round 2, all correct nodes will receive at least $n - f$ $(init', I_0, m, 1)$ messages, and they will all send $(echo', I_0, m, 1)$ message to all. Hence, at the end of the second phase of round 2 every correct node received $n - f$ $(echo', I_0, m, 1)$ messages, and therefore it accepts $(I_0, m, 1)$ at round 2.

If p accepted $(I_0, m, 1)$ in round 2, then p received $n - f$ $(echo', I_0, m, 1)$ messages. Hence, at least $n - 2 \cdot f$ such messages were sent by correct nodes in the previous phase of the that round. Therefore, at the first phase of round 3, all correct nodes will receive $n - 2 \cdot f$ $(echo', I_0, m, 1)$ messages and will all send $(echo', I_0, m, 1)$ (if

they haven't send such a message yet). Therefore, at the second phase of round 3, all correct nodes will receive $n - f$ $(echo', I_0, m, 1)$ messages, and will accept $(I_0, m, 1)$.

If p accepted $(I_0, m, 1)$ in round $r \geq 3$ or above, then some correct node received $n - f$ $(echo', I_0, m, 1)$ messages. Hence, at least $n - 2 \cdot f$ such messages were sent by correct nodes in previous rounds. Therefore, at round r, all correct nodes will receive $n - 2 \cdot f$ $(echo', I_0, m, 1)$ messages and will all send $(echo', I_0, m, 1)$ at the first phase of round $r + 1$ (if they haven't send such a message yet). Therefore, at the second phase of round $r + 1$, all correct nodes will receive $n - f$ $(echo', I_0, m, 1)$ messages, and will accept $(I_0, m, 1)$.

And we have shown that for every round r, if a correct node accepts $(I_0, m, 1)$, then all correct nodes accept $(I_0, m, 1)$ at round $r + 1$ or earlier.

□

Claim A.4. *The* Detection of broadcasters *property holds.*

Proof. For $p \in P$, the proof from [Toueg et al. 1987] holds. We consider $p = I_0$. Hence, we need to show that if a correct node p accepts $(I_0, m, 1)$ at round 1 or later, then every correct node has $I_0 \in broadcasters$ at the end of round 2.

We consider the first correct node p to accept $(I_0, m, 1)$ at some round r. If $r = 1$ then p received $n - f$ $(echo, I_0, m, 1)$ messages at the second phase of round 1. Hence, all correct nodes received $n - 2 \cdot f$ such messages by the first phase of round 2, and therefore all correct nodes sent $(init', I_0, m, 1)$ at that phase. Hence, at the second phase of round 2, all correct nodes add I_0 to $broadcasters$.

If $r = 2$, then p received $n - f$ $(echo', I_0, m, 1)$ messages at phase 2 of round 2. Hence, it received at least one such message from a correct node p', and therefore, p' received $n - f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2. Hence $n - 2 \cdot f$ correct nodes sent $(init', I_0, m, 1)$. Therefore, all correct nodes received $n - 2 \cdot f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2, and all correct nodes add I_0 to $broadcasters$.

If $r \geq 3$, then p received $(echo', I_0, m, 1)$ from some correct node p' at some round. We consider the first such p'. p' must have received $n - f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2 (otherwise, it received $n - 2 \cdot f$ $echo'$ messages at phase 1 of round $r' \geq 3$ which means it received an $echo'$ message from a correct node from a round before r', in contradiction to p' being the first node to send $echo'$ message). Hence, all correct nodes received $n - 2 \cdot f$ $(init', I_0, m, 1)$ messages at phase 1 of round 2, and therefore, all correct nodes have $I_0 \in broadcasters$ at the end of round 2.

And we have shown that if a correct node accepts $(I_0, m, 1)$, then all correct nodes have $I_0 \in broadcasters$ at the end of round 2. □

A.2 Byz-Consensus – Proof of Correctness

Lemma A.5. *The "Agreement" condition holds for Byz-Consensus.*

Proof. First we show that messages sent as part of the *broadcast* primitive, by correct nodes, contain a single value v. That is, $echo, init'$ and $echo'$ are all sent with $m = v$.

Messages can be sent either due to a *broadcast* executed, or due to line 3.a. If a message is sent due to a *broadcast*, then it is sent since $v \neq \perp$, hence it is sent due to $v = v'$ in line 3.b or in line 4.b. In the later case, it can be sent only if $(I_0, v', 1)$ was accepted, which means that an $(echo, I_0, v', 1)$ was sent by some correct node. If it is due to $v = v'$ in line 3.a it also means that an $(echo, I_0, v', 1)$ was sent by some correct node.

We assume by contradiction that there are two different correct nodes, one sent $(echo, I_0, v_1, 1)$ and the other sent $(echo, I_0, v_2, 1)$. We note that in order for a correct node to send $(echo, I_0, v', 1)$, it must have received $n-f$ message in the first phase of round 1, with the same value v'. Hence, there must have been $n-2 \cdot f > \frac{n}{2}$ correct nodes with the same initial value. Therefore, more than half of the nodes had v_1 as their initial value, and more than half of the nodes had v_2 as their initial value. And we reached a contradiction.

An immediate conclusion from the above, is that if all correct nodes return a value other than $\perp$, then they all return the same value. Hence, all we need to show is, that either all correct nodes return $\perp$, or they all return some value other than $\perp$.

If all correct nodes return $\perp$, we're done. Otherwise, there is some correct node p, that returned some value $w \neq \perp$. We examine the node p that was first to return a value $\neq \perp$. In order for node p to return a value $\neq \perp$, it must have set $v = v'$ at some stage. We consider 2 options: v was set due to line 3.b, or it was set later.

If v was set by line 3.b, then p received $n - f$ $(echo, I_0, w, 1)$ messages at round 1. Therefore, according to the *broadcast* primitive, p accepts $(I_0, w, 1)$. Therefore, according to the *Relay* property of the *broadcast* primitive, we have that all correct nodes accept $(I_0, w, 1)$ in round 2. Moreover, p executed *broadcast* $(p, w, 2)$, and due to the *Correctness* property, all correct nodes accept $(p, w, 2)$ by round 2. Hence, when executing line 4.b, all correct nodes have accepted $(I_0, w, 1)$ and $(p, w, 2)$, therefore they all set $v := w$, and in the following loop step, they will all enter line 4.a, decide on w and stop.

We now consider the case where p set $v = w$ after line 3.b. Since $w \neq \perp$, v must have been updated by line 4.b (note that this update can only be done once on p, since afterwards (in the next loop step) p stops executing the protocol). We consider 2 options: v was updated on some round $r < f + 2$, or v was update on round $r = f + 2$. In the first case, v was update at node p on round $r \leq f + 1$. Therefore, p had accepted $(I_0, w, 1)$ and another $r-1$ messages of the form (p_i, w, i) for all $i, 2 \leq i \leq r$. Therefore, due to the *Relay* property, all correct nodes will accept $(I_0, w, 1)$ and (p_i, w, i) for all $i, 2 \leq i \leq r$ by next round. Also, since next round $v \neq \perp$ at p, p will enter line 4.a, and will *broadcast* $(p, w, r + 1)$, and due to the *Correctness* property, all correct nodes accept $(p, w, r + 1)$ at round $r + 1$. Therefore, at round $r+1$, all correct nodes have accepted $(I_0, w, 1)$ and (p_i, w, i) for all $i, 2 \leq i \leq r+1$, and therefore, all correct nodes set $v := w$. Note that all correct nodes don't stop on line 4.c, due to the *Detection of broadcasters* property. That is, because p accepted $r - 1$ $(\_, w, i)$ for all $1 \leq i \leq r - 1$, then each correct node has at least $r' - 1$ nodes in *broadcasters* by round r' for all $1 \leq r' \leq r$. Therefore, no correct node stopped due to line 4.c.

We now consider the last option, that is, that p updated $v := w \neq \perp$ at round

$f+2$. Therefore, p accepted $(I_0, w, 1)$ and (p_i, w, i) for all $i, 2 \leq i \leq f+2$. Because there are no more than f *Byzantine* nodes, we have that one of the $p_i = p'$ is a correct node. Due to the *Unforgeability* property, p' broadcasted (p', w, j) at some round j. In order for p' to broadcast, it must have set $v := w$ at some round. We choose p to be the first node to set $v := w$, and hence, p' cannot set $v := w$ at round $j < f+1$, and we reach a contradiction. Hence, either the first correct node to set its value $v := w$ does so in a round $r \leq f+1$, or it doesn't do so at all.

And we have shown that if some node returned $w \neq \perp$ then all correct nodes return w, otherwise, all correct nodes return $\perp$. □

LEMMA A.6. *The "Validity" condition holds for Byz-Consensus.*

PROOF. If all correct nodes have the same initial value v_0, then in phase 1, all correct nodes send (v_0). Therefore, at the end of phase 1, all correct nodes received $n-f$ distinct messages of the form (v_0). Hence, at phase 2, all correct nodes send $(echo, I_0, v_0, 1)$ to everyone. Therefore, at the end of phase 2, all correct nodes receive $n-f$ distinct messages of the form $(echo, I_0, v_0, 1)$. Hence, according to line 3.b, they all set $v = v_0$, and at the beginning of line 4.a, they stop and return the agreed value v_0.

And we have shown that if all correct nodes have the same initial value $v_0 \neq \perp$ then that is their output value. □

LEMMA A.7. *The "Termination" condition holds for Byz-Consensus.*

PROOF. Define $\Delta = 2 \cdot (f+2)$. Clearly, within Δ phases, all correct nodes terminate. This is because each round consists of two phases, and the main loop executes no more than $f+1$ times, in addition to the 2 phases executed outside of the loop. □

LEMMA A.8. *The "Solidarity" condition holds for Byz-Consensus.*

PROOF. There are two locations in which the output can be changed such that it is not $\perp$. Either in line 3.b, or in line 4.b. If at least one correct node returned a value v' due to execution of line 3.b, then it received $n-f$ $(echo, I_0, v', 1)$ messages. Hence, it received at least one such echo message from a correct node p'. Therefore, in the previous round, p' received at least $n-f$ (v') messages. Therefore, at least $n-2 \cdot f$ correct nodes send (v') which means that at least $n-2 \cdot f$ had v' as their initial value. And "Solidarity" holds.

If all correct nodes returned v' due to executing line 4.b, then it means that a correct node accepted $(I_0, v', 1)$. Therefore, it either received $n-f$ $(echo', I_0, v', 1)$ messages, or $n-f$ $(echo, I_0, v', 1)$ messages. In the first case, at least one correct node had to send $(echo', I_0, v', 1)$ message, which means that it received at least one $(init', I_0, v', 1)$ message from a correct node, which means that this node received at least one $(echo, I_0, v', 1)$ message from a correct node. And we have that in order for a correct node to accept $(I_0, v', 1)$, some correct node must have sent $(echo, I_0, v', 1)$ at some time. This message can be sent (by a correct node) only at line 3.a. Therefore, the correct node that sent it, received at least $n-f$ messages of the form (v'). This means that at least $n-2 \cdot f$ correct nodes had v' as their initial value. And "Solidarity" holds. □

REFERENCES

Arora, A., Dolev, S., , and Gouda, M. 1991. Maintaining digital clocks in step. *Parallel Processing Letters 1*, 11–18.

Daliot, A. and Dolev, D. 2005a. Self-stabilization of byzantine protocols. In *In Proc. of the 7th Symposium on Self-Stabilizing Systems (SSS'05)*. Barcelona, Spain.

Daliot, A. and Dolev, D. 2005b. Self-stabilizing byzantine pulse synchronization. Technical report. Aug. url: http://arxiv.org/abs/cs.DC/0608092.

Daliot, A. and Dolev, D. 2006. Self-stabilizing byzantine agreement. In *In Proc. of the Twenty-fifth ACM Symposium on Principles of Distributed Computing (PODC'06)*. Denver, Colorado.

Daliot, A., Dolev, D., and Parnas, H. 2003a. Linear time byzantine self-stabilizing clock synchronization. In *Proc. of 7th Int. Conference on Principles of Distributed Systems (OPODIS'03)*. La Martinique, France. A corrected version appears in http://arxiv.org/abs/cs.DC/0608096.

Daliot, A., Dolev, D., and Parnas, H. 2003b. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In *In Proceedings of the Sixth Symposium on Self-Stabilizing Systems (DSN SSS '03)*. LNCS 2704. San Francisco.

Dolev, D. 1982. The byzantine generals strike again. *Journal of Algorithms 3*, 14–30.

Dolev, S. 1997. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Journal of Real-Time Systems 12,* 1, 95–107.

Dolev, S. and Welch, J. L. 1997. Wait-free clock synchronization. *Algorithmica 18,* 4, 486–511.

Dolev, S. and Welch, J. L. 2004. Self-stabilizing clock synchronization in the presence of byzantine faults. *Journal of the ACM 51,* 5, 780–799.

Herman, T. 2000. Phase clocks for transient fault repair. *IEEE Transactions on Parallel and Distributed Systems 11,* 10, 1048–1057.

Liskov, B. 1991. Practical use of synchronized clocks in distributed systems. In *Proceedings of 10th ACM Symposium on the Principles of Distributed Computing*.

Malekpour, M. R. and Siminiceanu, R. 2006. Comments on the byzantine self-stabilizing pulse synchronization protocol: Counterexamples. Technical Memorandum NASA-TM213951, NASA. Feb. http://hdl.handle.net/2002/16159.

Papatriantafilou, M. and Tsigas, P. 1997. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters 7,* 3, 321–328.

Toueg, S., Perry, K. J., and Srikanth, T. K. 1987. Fast distributed agreement. *SIAM Journal on Computing 16,* 3 (Jun), 445–457.

Received February 2007; accepted ????