Fast Self-Stabilizing Byzantine Tolerant Digital Clock Synchronization*

Michael Ben-Or‡
Hebrew University
benor@cs.huji.ac.il

Danny Dolev
Hebrew University
dolev@cs.huji.ac.il

Ezra N. Hoch
Hebrew University
ezraho@cs.huji.ac.il

ABSTRACT

Consider a distributed network in which up to a third of the nodes may be *Byzantine*, and in which the non-faulty nodes may be subject to transient faults that alter their memory in an arbitrary fashion. Within the context of this model, we are interested in the digital clock synchronization problem; which consists of agreeing on bounded integer counters, and increasing these counters regularly.

It has been postulated in the past that synchronization cannot be solved in a *Byzantine* tolerant and self-stabilizing manner. The first solution to this problem had an expected exponential convergence time. Later, a deterministic solution was published with linear convergence time, which is optimal for deterministic solutions.

In the current paper we achieve an expected constant convergence time. We thus obtain the optimal probabilistic solution, both in terms of convergence time and in terms of resilience to *Byzantine* adversaries.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Reliability, Theory

Keywords

Distributed computing, fault tolerance, self-stabilization, Byzantine failures, clock synchronization, digital clock synchronization.

*To appear in PODC'08, August, Toronto.
†This work was funded in part by Israel Science Foundation.
‡Incumbent of the Jean and Helena Alfassa Chair in Computer Science.

1. INTRODUCTION

Clock synchronization is a fundamental building block of distributed systems. The vast majority of distributed tasks require some sort of synchronization; clock synchronization is a very straightforward and intuitive tool for supplying this. Thus, it is desirable to have a highly robust clock synchronization mechanism available. A typical self-stabilizing algorithm seeks to re-establish synchronization once lost; a *Byzantine* tolerant algorithm assumes synchronization is never lost and focuses on containing the influence of the permanent presence of faulty nodes. The robustness we provide achieves both synchronization and *Byzantine* tolerance.

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 term "beat" in order to stay clear of any confusion with "pulse synchronization" or "clock ticks"). The digital clock synchronization problem seeks to ensure that all correct nodes eventually hold the same value for 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" at each beat.

Clock synchronization in a similar model has earlier been denoted as "digital clock synchronization" (see [1, 8, 16]) or "synchronization of phase-clocks" (see [14]); we will simply use the term "clock synchronization". However, these solutions do not consider *Byzantine* adversaries. In recent years there has been major advancement in self-stabilizing digital clock synchronization algorithms that are *Byzantine* tolerant. Starting from [9] which gives an expected exponential convergence time, and continuing with [7] which has deterministic linear convergence time. The current work continues this line, by providing a solution with expected constant convergence time.

In the classical "*Byzantine* field", it was shown that deterministic *Byzantine* agreement protocols require linear running time (see [13]). Randomization can be used to break this linear-time barrier, for example see [2], [4], [3] and [12]. The main idea behind such algorithms is to agree on a common coin with some probability. Each time all non-faulty nodes have the same random bit, they have a certain probability of reaching an agreement (on the input values).

The structure of such agreement protocols implies that their expected convergence time depends highly on the probability p that the common-coin algorithm will produce the same

coin at all non-faulty nodes; specifically, their expected convergence time is $O(\frac{1}{p})$.

Our solution can use any common-coin protocol[1] operating in a synchronous network, provided that the protocol has a constant probability of producing the same random bit at all non-faulty nodes upon termination (such as the protocol described in [12]). By re-executing this protocol anew each round, a "stream" of random bits is produced - one bit each round. Combining this with a single round "agreement phase" produces the basis for our self-stabilizing *Byzantine* tolerant clock synchronization.

The main contribution of the paper is a self-stabilizing, *Byzantine*-tolerant digital clock synchronization algorithm that converges in expected constant time; the algorithm is optimal in its convergence time and in its resiliency (it is resilient to $f < \frac{n}{3}$ *Byzantine* nodes.)

Aside from the clock synchronization algorithm, we show how to turn a non-self-stabilizing common-coin algorithm into a self-stabilizing one that produces a random bit every round.

1.1 Previous Self-stabilizing Byzantine Tolerant Clock Synchronization Algorithms

Previously, two different models were considered within the scope of the self-stabilizing *Byzantine* tolerant clock synchronization problem. The Global Beat System model (also known as the "synchronous" model), and the Bounded-Delay model (also known as the "semi-synchronous" model). In the synchronous model there is some device which is connected to all nodes and simultaneously delivers a signal to all nodes regularly. The semi-synchronous model assumes a bound on the message delivery time between two nodes.

The clock synchronization problem is slightly different in the different models: in the synchronous model, all nodes have an integer counter, and all non-faulty nodes must agree on the counter's value and increment it every time they receive a signal from the global device; this problem is sometimes referred to as "*digital* clock synchronization". In the semi-synchronous model, each node is equipped with a physical clock that can only measure the passing of time (but cannot tell the current time). All non-faulty nodes' physical clocks advance (more or less) at the same rate. In this setting, the clock synchronization problem consists of all non-faulty nodes having clock variables s.t. the difference between any two non-faulty nodes' clocks is bounded. Clearly, it is easier to solve the clock synchronization problem in the synchronous model.

The self-stabilizing *Byzantine*-tolerant clock synchronization problem was first tackled in [10] (see [9] for the full version). It was solved in both models using probabilistic algorithms, with expected exponential convergence time. The *Byzantine* resiliency supported by [10] is optimal.

Later on, a series of papers addressed the clock synchronization problem in both models using deterministic algorithms.

[1]The common coin protocol cannot rely on special initialization of all non-faulty nodes, such as assumed in [17].

In the bounded-delay model the first deterministic polynomial solution was presented in [5]. The polynomial convergence was obtained using a pulse synchronization protocol as a building block.[2] [6] provides an optimal (deterministic) solution in terms of convergence time and *Byzantine* resiliency for the bounded delay model, using a much simpler pulse producing protocol.

In the synchronous model, [15] solved the digital clock synchronization problem with deterministic linear convergence time. The main drawback of [15] is its *Byzantine*-resiliency, which was limited to $f < \frac{n}{4}$, as opposed to the optimal $f < \frac{n}{3}$. The result of [7] supports $f < \frac{n}{3}$ *Byzantine* nodes, while maintaining the deterministic linear convergence time. In [7] the clock synchronization problem is solved via an underlying "pulse algorithm" (see [7] for more information).

In the current work we solve the digital clock synchronization problem in the synchronous model, in a probabilistic manner, with expected constant convergence time. This solution is optimal in terms of its convergence time and in terms of its *Byzantine* resiliency.

Table 1 presents a summary of previous results as compared to the current paper.

Table 1: Summary of previous results

Paper	Model	Convergence	Resiliency
[10]	sync, probabilistic	$O(2^{2(n-f)})$	$f < \frac{n}{3}$
[10]	semi-sync, probabilistic	$O(n^{6(n-f)})$	$f < \frac{n}{3}$
[15]	sync, deterministic	$O(f)$	$f < \frac{n}{4}$
[7]	sync, deterministic	$O(f)$	$f < \frac{n}{3}$
[6, 5]	semi-sync, deterministic	$O(f)$	$f < \frac{n}{3}$
current	sync, probabilistic	$O(1)$	$f < \frac{n}{3}$

The paper is structured as follows: in Section 2 the model is presented along with a self-stabilizing coin-flipping protocol. Section 3 defines the k-Clock problem and solves it for $k = 2$, Section 4 solves the 4-Clock problem, Section 5 includes a solution to the k-Clock problem for any k; and lastly, Section 6 concludes with a discussion of the results.

2. MODEL AND DEFINITIONS

Our model consists of a distributed network of n nodes, which are fully connected to each other. Nodes communicate via message passing, and are all connected to a global beat system that provides "beats" at regular intervals; each beat reaches all nodes simultaneously. Each message m that is sent from p to q at beat r is guaranteed to reach q before beat $r+1$. In the following discussion, the term "round" will be used in the context of non-self stabilizing synchronous algorithms, and "beat" will be used when talking about self-stabilizing synchronous algorithms. Notice that "beat" can be used to denote the signal received from the global beat system, as well as the interval between such two signals; when not stated otherwise, "beat" will refer to the second meaning.

[2]The pulse synchronization protocol in the original version of [5] had a flaw, but any other pulse synchronization protocols can be used, as appears in the corrected version there.

A percentage of the nodes may be *Byzantine*, and behave arbitrarily; the presented algorithms are resilient to $f < \frac{1}{3} \cdot n$ such faulty nodes. We assume an information theoretic adversary with private channels. That is, the *Byzantine* adversary has access to all communications between faulty and non-faulty nodes; however, it does not have access to communications among non-faulty nodes. Moreover, the non-faulty nodes cannot use any computational assumptions (e.g. signatures) to guard against the adversary.

In addition to the faulty nodes, non-faulty nodes may undergo transient faults that change their memory in an arbitrary manner. Any resilient protocol is thus required to converge from any memory state. More specifically, following a long-enough period without any new transient faults, the system is required to converge to a state in which all correct nodes have synchronized digital clocks.

DEFINITION 2.1. *A node is* **non-faulty** *when it follows the given protocol. A node is* **faulty** *if it violates its protocol in any way. The terms Faulty and* Byzantine *will be used interchangeably.*

At times of transient failures one cannot assume anything about the state of any node, and the communication network may also behave erratically. Eventually the system becomes coherent again. In such a situation the system may find itself in an arbitrary state.

DEFINITION 2.2. *The communication network is* **non-faulty** *when the following conditions hold:*

1. *A message by a correct node p sent upon a receipt of a beat from the global beat system, arrives (and is processed) at its destination before the following beat is issued by the global beat system;*
2. *The sender's identity and the message context of any message received are not tampered with.*
3. *A message received by p was sent by some node no more than one beat ago. That is, "phantom" messages are not delivered.*

In real-world networks, it may take some time for the communication network to overcome transient faults. Specifically, the communication networks' buffers may contain messages that were not recently sent by any currently operating node, and the network may eventually deliver them. We consider the communication network to be *non-faulty* only after all of these "phantom" messages have been delivered or cleared away.

According to the above definition, once the network is non-faulty, it adheres to the global-beat-system model. Which means that messages cannot be lost and old messages cannot be stored for an arbitrarily long time.

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

DEFINITION 2.3. *A non-faulty node is considered* **correct** *only if it remains non-faulty for Δ_{node} beats during which the entire communication network is non-faulty.*[3]

Intuitively, for the system to converge to its desired state, it is required that a "critical mass" of non-faulty nodes have been clear of transient faults for a "long enough" period of time.

DEFINITION 2.4. *The system is* **coherent** *when the communication network is non-faulty and there are $n-f$ correct nodes.*

DEFINITION 2.5. *A beat interval $T = [r_1, r_2]$ is a* **coherent** *beat interval if during T the system is coherent and there is a set of the same $n-f$ nodes that are correct throughout T.*[4]

Note that in the above definition, the set of non-faulty nodes may not change from beat to beat. Alternatively, we could require that at each beat $r \in T$ there must be $n-f$ correct nodes, but they do not need to be the same correct nodes each beat. However, such a definition would complicate the proofs. The algorithms presented below are valid under both definitions; for the sake of clarity, the stronger assumption is used.

REMARK 2.1. *The values of n and f are fixed constants and are considered part of the "code" and therefore non-faulty nodes cannot initialize with arbitrary values for these constants.*

2.1 Common Coin-Flipping

Our clock synchronization algorithm uses a common coin-flipping (coin-flipping for short) algorithm as a building block. A coin-flipping algorithm is a distributed algorithm that, with some constant probability, produces an output bit that is common to all non-faulty nodes. Different coin-flipping algorithms exist (see [12] and [11]), with different properties. Our formalization and requirements of a coin-flipping algorithm are as follows:

DEFINITION 2.6. *An algorithm $\mathcal{A}$ is said to be a probabilistic coin-flipping algorithm if $\mathcal{A}$ has the following properties:*
*(***model***): $\mathcal{A}$ operates in a synchronous network, communicates only via message passing, and is resilient to $f < \frac{n}{3}$* Byzantine *nodes;*
*(***termination***): There exists a constant $\Delta_{\mathcal{A}}$, such that $\mathcal{A}$ terminates within $\Delta_{\mathcal{A}}$ rounds of sending-and-receiving messages;*
*(***binary output***): The output of $\mathcal{A}$ at each node i is d_i, $d_i \in \{0, 1\}$;*

[3]The assumed value of Δ_{node} in the current paper will be defined later.
[4]The indexing of the beats is not available to the nodes, it is used only for presentation purposes.

Algorithm SS-BYZ-COIN-FLIP /* executed at each node, each beat */
/* $\mathcal{A}$ is a probabilistic coin-flipping algorithm */
/* the A_i's are $\Delta_{\mathcal{A}}$ instances of $\mathcal{A}$ */

On beat (signal from global beat system):

1. **For** $i := 1$ to $\Delta_{\mathcal{A}}$
   execute the ith round of A_i;
2. **Output** the value of $A_{\Delta_{\mathcal{A}}}$;
3. **For** $i := 1$ to $\Delta_{\mathcal{A}} - 1$
   $A_{i+1} := A_i$;
4. **Initialize** A_1 to be a new instance of $\mathcal{A}$;

Figure 1: A self-stabilizing coin-flipping algorithm.

*(**event** E_0): The event that all non-faulty nodes have the same output "0", occurs with constant probability $p_0 > 0$;*
*(**event** E_1): The event that all non-faulty nodes have the same output "1", occurs with constant probability $p_1 > 0$;*
*(**unpredictability**): If either E_0 or E_1 occurs, then the probability of any f nodes to predict the output of $\mathcal{A}$ by the end of round $\Delta_{\mathcal{A}} - 1$ is no more than $1 - \min\{p_0, p_1\}$.*

Intuitively, when executing a probabilistic coin-flipping algorithm $\mathcal{A}$ there is a constant probability that all non-faulty nodes have the same output value. Moreover, the *Byzantine* nodes do not "know" which of the possible outputs will be the common output until the very last round.

REMARK 2.2. *The probability space of the above definition is valid for any choice of* Byzantine *adversary. That is, an algorithm $\mathcal{A}$ is a probabilistic coin-flipping algorithm, if for any* Byzantine *adversary, the properties of Definition 2.6 hold regarding all possible runs. (for more details see [12]).*

In the following section, a probabilistic coin-flipping algorithm $\mathcal{A}$ is assumed to be "self-contained", in the sense that multiple invocations of $\mathcal{A}$ do not affect the probability of the events E_0 or E_1 occurring within each invocation, or the probability of *Byzantine* nodes predicting the output before round $\Delta_{\mathcal{A}}$ of each invocation. This "self-contained"-ness is required to allow multiple concurrent executions of $\mathcal{A}$ to run properly.

Ensuring such "self-contained"-ness could be a problem in asynchronous systems. However, it is easy to implement in the global-beat-system model when each instance of $\mathcal{A}$ terminates within a finite number of rounds: simply add a "session number" to each instance of $\mathcal{A}$, and differentiate messages of co-executing instances using this session number. Since only a finite number of instances are concurrently executed at any round, the session numbers can be "recycled", thus avoiding problems of infinite counters in the setting of self-stabilization.

OBSERVATION 2.1. *The common coin protocol of [12] adheres to Definition 2.6. In [12] the common coin protocol is based on graded verifiable secret sharing (GVSS), which has 3 "phases":* share, decide, recover. *Up until the last phase, the secret is unrecoverable by any set of f or less nodes. Moreover, the* recover *phase is one round long. Thus, the "*unpredictability*" property of Definition 2.6 holds.*

[12]'s common coin protocol executes a GVSS protocol for each node in the system. However, the last step of the common coin protocol consists of executing the recover *phase of all the GVSS instances, which conserves the property that the output of the common coin is unpredictable by any set of $\leq f$ nodes, until the very last round.*

Lastly, the protocols of [12] operate in a synchronous model and are tolerant to $f < \frac{n}{3}$ Byzantine *nodes. Moreover, the adversarial model assumed in [12] allows "rushing"; thus, when using [12], our solution is also tolerant to rushing.*

REMARK 2.3. *The protocol of [12] requires the values of n, f as input, as well as additional constants; for example, the secret sharing protocol of [12] requires a prime $p > n$. These constants are assumed to be part of the "code" and non-faulty nodes do not initialize with arbitrary values of these constants.*[5]

2.2 Self-stabilizing Coin-flipping

When considering a system that is self-stabilizing, round numbers become a problematic notion, since different nodes may have different values as their current "round number". Thus, statements such as "$\mathcal{A}$ terminates within $\Delta_{\mathcal{A}}$ rounds" require some explanation. To this end, we define pipelined probabilistic coin-flipping, and later use it to define a self-stabilizing coin-flipping algorithm.

DEFINITION 2.7. *An algorithm $\mathcal{B}$ is said to be a pipelined probabilistic coin-flipping algorithm if $\mathcal{B}$ has the following properties:*
*(**model**): $\mathcal{B}$ operates in a synchronous network, communicates only via message passing, and is resilient to $f < \frac{n}{3}$* Byzantine *nodes;*
*(**binary output**): Each round, the output of $\mathcal{B}$ at each node i is d_i, $d_i \in \{0, 1\}$;*
*(**event** E_0): Each round, the event that all non-faulty nodes have the same output value "0", occurs with constant probability $p_0 > 0$;*

[5] These constants can be computed in a single way given the value of n (for example, let p be the smallest prime that is larger than n). Thus, this assumption does not weaken the result.

*(**event** E_1): Each round, the event that all non-faulty nodes have the same output value "1", occurs with constant probability $p_1 > 0$;*
*(**unpredictability**): The probability that either E_0 or E_1 occurs at some round is independent of the previous rounds. If either E_0 or E_1 occurs, then the probability that any f nodes will predict the output of $\mathcal{B}$ at round r by the end of round $r-1$ is no more than $1 - \min\{p_0, p_1\}$.*

REMARK 2.4. *The "unpredictability" property means that as far as the adversary can know (considering all information it has access to) the output of the random bit at each round is independent of previous rounds. However, this is not the usual meaning of "independent", as if the adversary has all the information of all correct nodes, the random bits can be predicted. (see [12] for more information).*

In the rest of this paper we use the term "independent" to mean "as far as the adversary can tell, the events are independent".

Informally, the above definition states that at every round, with constant probability, all non-faulty nodes agree on a common random bit.

DEFINITION 2.8. *An algorithm $\mathcal{C}$ is said to be a self-stabilizing probabilistic coin-flipping algorithm if $\mathcal{C}$ has the following properties:*
*(**model**): $\mathcal{C}$ operates in a self-stabilizing synchronous network (i.e. with a global beat system), communicates only via message passing, and is resilient to $f < \frac{n}{3}$ Byzantine nodes;*
*(**convergence**): Starting from any arbitrary state, $\mathcal{C}$ converges within $\Delta_{\mathcal{C}}$ beats to be a pipelined probabilistic coin-flipping algorithm.*

Given an algorithm $\mathcal{A}$ that is a probabilistic coin-flipping algorithm, one can construct an algorithm $\mathcal{C}$ that is a self-stabilizing probabilistic coin-flipping algorithm. See Figure 1.

LEMMA 1. *Given a probabilistic coin-flipping algorithm $\mathcal{A}$, the algorithm* SS-BYZ-COIN-FLIP *is a self-stabilizing coin-flipping algorithm, with convergence time $\Delta_{\text{SS-BYZ-COIN-FLIP}} = \Delta_{\mathcal{A}}$.*

PROOF. Consider a system that has been coherent for $\Delta_{\mathcal{A}}$ beats, and a set of $n-f$ non-faulty nodes $\mathcal{G}$, where each node has been non-faulty for $\Delta_{\mathcal{A}}$ beats. The nodes in $\mathcal{G}$, when executing Line 2, output the value of a probabilistic coin-flipping algorithm that has been initialized and executed properly for $\Delta_{\mathcal{A}}$ rounds, and therefore its properties hold. This situation continues to hold for as long as the nodes in $\mathcal{G}$ are not subject to transient faults. SS-BYZ-COIN-FLIP therefore converges, within $\Delta_{\mathcal{A}}$ beats, to become a pipelined probabilistic coin-flipping algorithm. □

THEOREM 1. *There exists a self-stabilizing probabilistic coin-flipping algorithm, with constant stabilization time.*

PROOF. Denote the coin-flipping algorithm in [12] by $\mathcal{OC}$; $\mathcal{OC}$ has the properties of a probabilistic coin-flipping algorithm, as defined in Definition 2.6 (see Observation 2.1). Thus, the algorithm SS-BYZ-COIN-FLIP (when executed with $\mathcal{A} := \mathcal{OC}$) is a self-stabilizing probabilistic coin-flipping algorithm, according to Lemma 1. In addition, $\Delta_{\mathcal{OC}}$ is constant, leading to a constant stabilization time of SS-BYZ-COIN-FLIP, as required. □

3. THE DIGITAL CLOCK SYNCHRONIZATION PROBLEM

In the digital clock synchronization problem each node u has an integer variable $u.clock$ representing the node's clock-value. The goal is to synchronize all correct nodes' *clock* variables.

DEFINITION 3.1. *A system is* **clock-synched** *at beat r with value $Clock(r)$, if at the end of beat r, all correct nodes have the same clock-value, and it is equal to $Clock(r)$.*

DEFINITION 3.2. *The $k-Clock$ problem consists of the following:*
(convergence) starting from any state, eventually (at some beat r) the system becomes clock-synched with value $Clock(r)$;
(closure) from this point on the system stays clock-synched s.t. at beat $r+i$ it is clock-synched with value $Clock(r)+i \bmod k$.

3.1 Overview of the Solution

The first step is to construct a 2-Clock algorithm SS-BYZ-2-CLOCK using the self-stabilizing coin-flipping algorithm SS-BYZ-COIN-FLIP. Then, by using 2 instances of SS-BYZ-2-CLOCK, a 4-clock algorithm SS-BYZ-4-CLOCK is built. Using SS-BYZ-4-CLOCK one can have four send-and-receive "phases" before a wrap-around of the clock value occurs. Using SS-BYZ-4-CLOCK, SS-BYZ-CLOCK-SYNC is constructed, which runs SS-BYZ-4-CLOCK, and sends messages each time SS-BYZ-4-CLOCK's clock value changes. Thus, between two wraparounds of SS-BYZ-4-CLOCK's clock it is possible to try to achieve an agreement on the clock value of SS-BYZ-CLOCK-SYNC, with a constant probability of success. This allows SS-BYZ-CLOCK-SYNC to solve the k-Clock problem for any k, in an expected constant number of beats.

Observe that each algorithm uses the previous algorithms as building blocks. On a beat received from the global-beat-system, each algorithm performs a step in each of the appropriate building blocks. We call such a step "execution of a single beat" of the relative algorithm.

3.2 Solving the 2-Clock Problem

Let $\mathcal{C}$ be a self-stabilizing probabilistic coin-flipping algorithm. At each beat, $\mathcal{C}$ produces some random bit. In SS-BYZ-2-CLOCK (see Figure 2), $\mathcal{C}$ is executed in the background, and each beat *rand* holds the random output bit of $\mathcal{C}$. The algorithm SS-BYZ-2-CLOCK requires that $\Delta_{node} \geq \Delta_{\mathcal{C}}$.

REMARK 3.1. *Consider some beat r. Notice that in the algorithm of Figure 2 the value of* rand *at beat r is used to "replace" $\perp$ values sent in beat $r-1$. One may try to use the*

Algorithm SS-BYZ-2-CLOCK /* executed at node u each beat */
/* $\mathcal{C}$ is self-stabilizing probabilistic coin-flipping algorithm */

On beat (signal from global beat system):

1. **broadcast**[a] $u.clock$; /* $u.clock \in \{0, 1, \perp\}$ */
2. **execute** a single beat of $\mathcal{C}$, and set *rand* to be the output of $\mathcal{C}$;
3. **consider** each message with "$\perp$" as carrying the value *rand*; /* $rand \in \{0, 1\}$ */
4. **set** maj to be the value that appeared the most, and $\#maj$ the number of times it appeared; /* $maj \in \{0, 1\}$ */
5. **if** $\#maj \geq n - f$ then $u.clock := 1 - maj$;
6. **else** $u.clock := \perp$;

[a]In the context of this paper, "broadcast" means "send the message to all nodes". (We **do not** assume broadcast channels.)

Figure 2: An algorithm that solves the 2-Clock problem.

value of rand *at beat* $r-1$*, and have each node send "rand" instead of "$\perp$" (during beat* $r-1$*). The problem with such a solution is that the* Byzantine *nodes can "decide" which value they send at beat* $r-1$ *according to the result of* rand *at beat* $r-1$*. This way we lose the power of randomization, since the* Byzantine *nodes' action can depend on the value of the random bit.*

To avoid this, rand *of beat* r *is used only after all nodes (including the* Byzantine *ones) sent their messages in beat* $r-1$*. That is,* rand *is used only after the* Byzantine *nodes are committed to the values they sent. Thus,* rand *is independent of the clock values sent by* Byzantine *nodes.*

OBSERVATION 3.1. *Consider two vectors* $\vec{A}, \vec{B}$ *of length* n *that differ in at most* f *entries, where* $n > 3f$*. If* $\vec{A}$ *contains* $n-f$ *copies of* v_A*, and* $\vec{B}$ *contains* $n-f$ *copies of* v_B*, then* $v_A = v_B$*.*

DEFINITION 3.3. *Let* T *be a coherent beat interval. For any* $r \in T$*:*
$clocks_r^{start}$ *is the set of all clock values of correct nodes at the beginning of beat* r*.*
$clocks_r^{end}$ *is the set of all clock values of correct nodes at the end of beat* r*.*

Let $\mathcal{G}$ be the set of correct nodes during beat $r \in T$; recall that $\mathcal{G}$ does not change throughout T (see Definition 2.5). $clocks_r^{start} := \{u.clock \mid u \in \mathcal{G}\}$ before the execution of Line 1, and $clocks_r^{end} := \{u.clock \mid u \in \mathcal{G}\}$ after the execution of Line 6. Notice that $clocks_r^{end} = clocks_{r+1}^{start}$. Note also that the system is clock-synched at beat r with value $v \in \{0, 1\}$ if (and only if) $clocks_r^{end} = \{v\}$.

LEMMA 2. *Let* T *be a coherent interval. If at some beat* $r \in T$*,* $clocks_r^{start} = \{v\}$*, (where* $v \in \{0, 1\}$*), then* $clocks_r^{end} = \{1 - v\}$*.*

PROOF. If $clocks_r^{start} = \{v\}$, then there are $n-f$ correct nodes at the beginning of beat r with $clock = v$; when they execute Line 1, they all send the same value v. Since $v \in \{0, 1\}$, each correct node receives at least $n-f$ messages with the same value v (see Observation 3.1), therefore $maj = v$ and $\#maj \geq n - f$. Thus, in Line 5, all correct nodes set $clock := 1 - maj = 1 - v$. □

DEFINITION 3.4. *A beat* r *is called "safe" if all correct nodes have the same value of* rand *during* r*.*

LEMMA 3. *Let* T *be a coherent interval. If* $r \in T$ *is a safe beat, then* $clocks_r^{end} \subset \{v, \perp\}$ *for* $v \in \{0, 1\}$*.*

PROOF. Correct nodes set *clock* either in Line 5 or in Line 6. Those that set *clock* in Line 6 set it to "$\perp$". Consider all correct nodes that set *clock* at Line 5; we show that they all set *clock* to the same value v. r is a safe beat, therefore, all correct nodes that sent "$\perp$" in Line 1 will be considered to have sent the same value by all correct nodes. Thus, two correct nodes can differ by at most f values when setting maj and $\#maj$ in Line 4. By Observation 3.1, all nodes that have $\#maj \geq n - f$ have the same value for maj. Thus, all nodes that update *clock* in Line 5 update it to the same value. □

LEMMA 4. *Let* T *be a coherent interval. If* $r \in T$ *is a safe beat in which* $clocks_r^{start} \subset \{v, \perp\}$ *for* $v \in \{0, 1\}$*, then with probability at least* $min\{p_0, p_1\}$*,* $clocks_r^{end} = \{v'\}$ *for* $v' \in \{0, 1\}$*.*

PROOF. If $clocks_r^{start} = \{v\}$ for $v \in \{0, 1\}$, then, by Lemma 2 we are done. Otherwise assume that $clocks_r^{start} \neq \{v\}$ for $v \in \{0, 1\}$.

r is a safe beat, therefore, all correct nodes have the same value of *rand*. Consider two cases: $clocks_r^{start} = \{\perp\}$ and $clocks_r^{start} \neq \{\perp\}$. In the first case, all nodes consider (in Line 3) to have received at least $n - f$ messages with value *rand*; thus they set $\#maj \geq n - f$ and $maj = rand$, and therefore, (after Line 5) $clocks_r^{end} = \{1 - rand\}$.

In the second case, $clocks_r^{start} \neq \{\perp\}$; thus, under the lemma's assumption we have that $clocks_r^{start} = \{v, \perp\}$ for $v \in \{0, 1\}$. Recall that $clocks_{r-1}^{end} = clocks_r^{start}$, thus, the

Algorithm SS-BYZ-4-CLOCK /* *executed at node u each beat* */
/* $\mathcal{A}_1, \mathcal{A}_2$ *are instances of* SS-BYZ-2-CLOCK */

On beat (signal from global beat system):

1. **execute** a single beat of $\mathcal{A}_1$;
2. **if** $u.clock(\mathcal{A}_1) = 0$ then
   execute a single beat of $\mathcal{A}_2$;
3. **set** $u.clock := 2 \cdot u.clock(\mathcal{A}_2) + u.clock(\mathcal{A}_1)$[a];

[a]To differentiate between the output *clock* value of SS-BYZ-4-CLOCK and that of SS-BYZ-2-CLOCK, consider $u.clock$ to be the output of SS-BYZ-4-CLOCK, $u.clock(\mathcal{A}_1)$ is the output of $\mathcal{A}_1$ and $u.clock(\mathcal{A}_2)$ is the output of $\mathcal{A}_2$.

Figure 3: An algorithm that solves the 4-Clock problem.

values of $clocks_r^{start}$ have been determined in beat $r-1$. The "unpredictability" property implies that *rand* is independent of "what happened" during beat $r-1$ (see Remark 3.1). We thus conclude that with probability at least $min\{p_0, p_1\}$, $rand = v$. In this case, all correct nodes have $\#maj \geq n - f$ and $maj = v$, thus all correct nodes have (after Line 5) $clocks_r^{end} = \{1 - v\}$.

Thus, with probability of at least $min\{p_0, p_1\}$ we have that $clocks_r^{end} = \{v'\}$ for $v' \in \{0, 1\}$. □

LEMMA 5. *Let T be a coherent interval. Any beat $r \in T$ is safe with probability $p_0 + p_1$.*

PROOF. Consider some beat $r \in T$; since T is coherent, there is a set of $n - f$ correct nodes during beat r. Since $\Delta_{node} \geq \Delta_{\mathcal{C}}$, they have all executed $\mathcal{C}$ for $\Delta_{\mathcal{C}}$ beats ($\mathcal{C}$'s required convergence time). Thus, properties "event E_0" and "event E_1" hold; which means that with probability p_0 all correct nodes have $rand = 0$ and with probability p_1 all correct nodes have $rand = 1$. Thus, with probability $p_0 + p_1$ the beat is safe. □

THEOREM 2. SS-BYZ-2-CLOCK *solves the 2-Clock problem with expected constant convergence time.*

REMARK 3.2. *When talking about the expected convergence time of* SS-BYZ-2-CLOCK, *it is convenient to think of an infinitely long coherent interval $T = [r, \infty]$. However, the above theorem holds also for short finite intervals, but would require saying: "T is of length of at least l, where at any beat after $r + l$ there is a constant probability that the algorithm converges". Instead, we simply say that T is "long enough".*

PROOF. Let $T = [r_1, r_2]$ be a "long enough" coherent interval. By Lemma 5, for each beat $r \in T$ there is a constant probability $c_1 := p_0 + p_1$ that r is a safe beat. By the "unpredictability" property, the probability of the event E that two consecutive beats $r, r+1$ are safe is at least c_1^2. From Lemma 3 and Lemma 4, given that E occurred, there is a probability of $c_2 = \min\{p_0, p_1\}$, and thus all correct nodes have the same *clock* value at the end of beat $r + 1$, and by Lemma 2, they continue to agree on it at the end of any beat $r' \geq r + 1, r' \in T$.

Thus, during each beat r, $r \in [r_1 + 1, r_2]$, there is a constant probability of $c_2 \cdot c_1^2$ that r is safe, and that $r - 1$ is safe, and that all correct nodes have the same *clock* value by the end of r. Therefore, after an expected constant number of beats (starting from $r_1 + 1$), all correct nodes agree on the *clock* value, and by Lemma 2, they all continue to agree on the *clock* value and change it from "1" to "0" and vice versa each beat. Hence, SS-BYZ-2-CLOCK solves the 2-Clock problem and has expected constant convergence time. □

Theorem 2 states that SS-BYZ-2-CLOCK converges with expected constant time. However, as can be seen by the proof of Theorem 2, the result is actually much stronger: if at some beat the algorithm has not yet converged, then it has a constant probability of converging in the next beat. Thus, denote by $\Delta_{\text{SS-BYZ-2-CLOCK}}$ the expected convergence time of SS-BYZ-2-CLOCK; the probability that SS-BYZ-2-CLOCK does not converge within $l \cdot \Delta_{\text{SS-BYZ-2-CLOCK}}$ beats decreases exponentially with l. Therefore, not only does SS-BYZ-2-CLOCK converge in expected constant time, it also does so with high probability.

4. SOLVING THE 4-CLOCK PROBLEM

The previous section solved the 2-Clock problem. The following describes how to solve the 4-Clock problem, using 2 instances of SS-BYZ-2-CLOCK, $\mathcal{A}_1$,and $\mathcal{A}_2$. The presented solution requires that $\Delta_{node} \geq \max\{\Delta_{\mathcal{A}_1}, 2 \cdot \Delta_{\mathcal{A}_2}\}$; since $\mathcal{A}_1, \mathcal{A}_2$ are both instances of SS-BYZ-2-CLOCK, Δ_{node} is set to be $2 \cdot \Delta_{\text{SS-BYZ-2-CLOCK}}$.

THEOREM 3. SS-BYZ-4-CLOCK *(Fig. Figure 3) solves the 4-Clock problem with expected constant convergence time.*

PROOF. Let $T = [r_1, r_2]$ be a "long enough"[6] coherent interval. A single beat of $\mathcal{A}_1$ is executed for every beat $r \in T$; thus $\mathcal{A}_1$ is executed properly by all correct nodes during T, and all lemmata regarding $\mathcal{A}_1$ hold. Therefore, $\mathcal{A}_1$ converges with expected constant time.

Let $r_{\mathcal{A}_1} \in T$ be the beat at which $\mathcal{A}_1$ has converged. Thus, for any beat $r \geq r_{\mathcal{A}_1}, r \in T$ all correct nodes alternate between executing a single beat of $\mathcal{A}_2$ and not executing $\mathcal{A}_2$. Thus, a single beat of $\mathcal{A}_2$ is executed every other beat in $[r_{\mathcal{A}_1}, r_2]$ by all correct nodes. Therefore, due to Theorem 2, $\mathcal{A}_2$ converges in expected constant time.

[6]See Remark 3.2.

```
Algorithm SS-BYZ-CLOCK-SYNC                          /* executed at node u each beat */
                                                     /* A solves the 4-Clock problem */
On beat (signal from global beat system):
  1. execute a single beat of A;
  2. set u.full_clock := u.full_clock + 1  mod k;
  3. case u.clock(A) of:                 /* consider u.clock(A) at the beginning of the beat */
     (a) u.clock(A) = 0: broadcast[a] u.full_clock to all;
     (b) u.clock(A) = 1: if received n − f same value v in previous beat
                             set propose := v;
                         else
                             set propose :=⊥;
                         fi
                         broadcast propose to all;
     (c) u.clock(A) = 2: let save be the majority value ≠⊥ of the received values;
                         (if all values are ⊥, set save :=⊥)
                         if save ≠⊥ appeared n − f times
                             set bit := 1;
                         else
                             set bit := 0;
                         fi
                         broadcast bit to all;
                         if save =⊥
                             set save := 0;
     (d) u.clock(A) = 3: if received n − f "1" in previous beat
                             set u.full_clock := save + 3;
                         else if received n − f "0" in previous beat
                             set u.full_clock := 0;
                         else if rand = 1
                             set u.full_clock := save + 3;
                         else if rand = 0
                             set u.full_clock := 0;
                         fi
```

[a] In the context of this paper, "broadcast" means "send the message to all nodes".

Figure 4: An algorithm that solves the k-Clock problem for any k.

Let $r_{\mathcal{A}_2}$, $r_{\mathcal{A}_2} \geq r_{\mathcal{A}_1}$ denote the beat at which $\mathcal{A}_2$ has converged. During the interval $[r_{\mathcal{A}_2}, r_2]$, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the following pattern: $(clock(\mathcal{A}_1) = 0, clock(\mathcal{A}_2) = 0), (clock(\mathcal{A}_1) = 1, clock(\mathcal{A}_2) = 0), (clock(\mathcal{A}_1) = 0, clock(\mathcal{A}_2) = 1), (clock(\mathcal{A}_1) = 1, clock(\mathcal{A}_2) = 1)$; this pattern continues until beat r_2. Thus, during the interval $[r_{\mathcal{A}_2}, r_2]$ the *clock* variable (that is updated in Line 3) has the following pattern: 0, 1, 2, 3; and this pattern continues until beat r_2.

To conclude, the linearity of expectations implies that $r_{\mathcal{A}_2} - r_1$ is constant in expectation. That is, SS-BYZ-4-CLOCK converges in expected constant time. □

REMARK 4.1. SS-BYZ-4-CLOCK *uses two different pipelines of coin-flipping. Actually, it could be improved to use a single coin-flipping pipeline, and reduce both the message complexity and expected convergence time. However, these improvements are only by a constant factor, and therefore are omitted for the sake of clarity.*

5. SOLVING THE K-CLOCK PROBLEM (FOR ANY K)

The construction of SS-BYZ-4-CLOCK can be similarly used to solve the 8-Clock problem from $\mathcal{A}_1$ that solves the 4-Clock problem and $\mathcal{A}_2$ that solves that 2-Clock problem. In general, any 2^{k+1}-Clock problem can be solved with $\mathcal{A}_1$ that solves 2^k-Clock and $\mathcal{A}_2$ that solves the 2-Clock problem. Even better, any $2^{2^{k+1}}$-Clock problem can be solved with $\mathcal{A}_1, \mathcal{A}_2$ that solve the 2^{2^k}-Clock problem. Thus, the k-Clock problem can be solved for infinitely many values. However, such constructions have $\log n$ overhead (or $\log \log n$, depending on which of the above schemas is used) in their message complexity and at least the same overhead in their expected convergence time.

The following SS-BYZ-CLOCK-SYNC (see Figure 4) algorithm has a constant overhead both in message complexity and in its expected convergence time. It uses a 4-Clock algorithm to construct a k-Clock algorithm for any value of k. The schema used is similar to the algorithm of Turpin and Coan (see [18]) when combined with the algorithm of Rabin (see [17]). More specifically, SS-BYZ-CLOCK-SYNC is constructed of 4 "phases", each executed in a consecutive beat. The first phase sends the clock value to everyone. In the second phase, each node votes on the majority clock value it received, or $\perp$ if no such value exists. The third phase determines whether enough nodes voted on a value $\neq\perp$, thus ensuring that those nodes that have voted, voted on the same value in phase 2.

Lastly, in the fourth phase the new clock value is set either to be the majority clock value of phase 2, or (if there were not enough votes) is randomly selected using the output of the coin-flipping algorithm.

In the context of SS-BYZ-CLOCK-SYNC, Δ_{node} is the same as in SS-BYZ-4-CLOCK.

$\mathcal{A}$ is an instance of SS-BYZ-4-CLOCK. The following discussion assumes that all correct nodes have the same value of $clock(\mathcal{A})$; it takes expected constant time until this happens. By this assumption, all correct nodes perform the same portion of the code on each beat. That is, they all perform Block 3.a on some beat, then on the next beat they perform Block 3.b, and so on. In the following lemmata, we assume that correct nodes operate in the following cycle: they all perform Block 3.a, the following beat they perform Block 3.b, then Block 3.c, then Block 3.d, and then they go back to performing Block 3.a.

DEFINITION 5.1. *Let T be a coherent interval; For any beat $r \in T$:*
$full_clocks_r^{start}$ is the set of all full_clock values of correct nodes at the beginning of beat r.
$full_clocks_r^{end}$ is the set of all full_clock values of correct nodes at the end of beat r.

Notice that $full_clocks_r^{start}, full_clocks_r^{end} \subset \{0, 1, \ldots, k-1\}$ for all r.

LEMMA 6. *Let T be a coherent interval and let $r \in T$ be a beat at which $clock(\mathcal{A}) = 3$; if $full_clocks_r^{end} = \{v\}$ then for every beat $r' > r, r' \in T$ it holds that*

$$full_clocks_{r'}^{start} = \left\{v + (r' - r - 1) \mod k\right\} .$$

PROOF. Assume the lemma holds for any beat r', $r \leq r' \leq r+5$. Recall that $full_clocks_{r+4}^{end} = full_clocks_{r+5}^{start}$. The assumption on r' implies that $full_clocks_{r+4}^{end} = \{v + 4 \mod k\}$. In addition, notice that at beat $r+4$, it holds that $clock(\mathcal{A}) = 3$. Now, repeatedly applying the above assumption leads to $full_clocks_{r+i}^{end} = \{v+i \mod k\}$ (for $i \geq 0, r+i \in T$). In other words, $full_clocks_{r'}^{end} = \{v + (r' - r) \mod k\}$ and $full_clocks_{r'}^{start} = \{v + (r' - r - 1) \mod k\}$. It is left to prove that the lemma holds for any beat r', $r \leq r' \leq r+5$.

For $r' = r+1$ the claim holds immediately; additionally, up until beat $r+5$, the correct nodes update *full_clock* only in Line 2. Therefore, they all update *full_clock* in the same way. Thus, $full_clocks_{r+1}^{start} = \{v\}$; $full_clocks_{r+2}^{start} = \{v+1\}$; $full_clocks_{r+3}^{start} = \{v+2\}$; $full_clocks_{r+4}^{start} = \{v+3\}$, (where "+" is modulo k).

It remains to show that $full_clocks_{r+5}^{start} = \{v+4\}$. We will do this by proving an equivalent claim, namely, that: $full_clocks_{r+4}^{end} = \{v+4\}$. To show this, consider the messages sent during beats $r+1, \ldots, r+4$.

At beat $r+1$ all correct nodes send $v+1$ to everyone, and so all correct nodes receive at least $n-f$ copies of $v+1$. At beat $r+2$ all correct nodes set $propose := v+1$. At beat $r+3$ all correct nodes have $save = v+1$ and they set $bit := 1$ and therefore all of them receive $n-f$ copies of "1". This implies that at beat $r+4$ all correct nodes set $full_clock := save + 3 = v + 1 + 3 = v + 4$. That is, $full_clocks_{r+4}^{end} = \{v+4\}$. □

LEMMA 7. *Let T be a coherent interval; At most one value $v \neq \perp$ can be sent in Block 3.b by correct nodes at some beat $r \in T$.*

PROOF. Immediate from Observation 3.1. □

LEMMA 8. *Let T be a coherent interval; If $r \in T$ is a safe beat at which $clock(\mathcal{A}) = 3$, then with probability at least $\min\{p_0, p_1\}$ all correct nodes have the same full_clock value.*

PROOF. First, consider the case in which no correct node sees $n-f$ copies of the same value. In this case, all correct nodes set $full_clock := 0$, with probability p_0.

If some correct node p receives $n-f$ copies of some value v, then all correct nodes that receive $n-f$ copies of some value, receive the same value v (by Observation 3.1). Notice that v is calculated according to messages determined at beat $r-1$, and *rand* was chosen at beat r. Therefore, due to "*unpredictability*", *rand* and v are independent of each other. Thus, with probability at least $\min\{p_0, p_1\}$, all correct nodes update *full_clock* in the same manner: either $full_clock := 0$ or $full_clock := save + 3$. If $v = 0$ then we are done. If $v = 1$ then we are left to show that all correct nodes have the same value of *save*.

Since p has received $n-f$ copies of "1" it means that some correct node q has sent "1" in beat $r-1$. Thus, q has received at least $n-f$ copies of $save_q \neq \perp$ at beat $r-2$. Thus, all other correct nodes have received at least $n-2f$ copies of $save_q$. By Lemma 7, correct nodes either sent $\perp$ or $save_q$ in beat $r-2$. Thus, correct nodes can receive at most $f < n-2f$ values that are not $\perp$ and are not $save_q$. Hence, all correct nodes have $save = save_q$. □

THEOREM 4. SS-BYZ-CLOCK-SYNC *solves the k-Clock problem for any value of k, and converges in expected constant time.*

PROOF. The proof is very similar to the proof of Theorem 2 and Theorem 3. By Lemma 8, after an expected constant number of beats all correct nodes have the same *full_clock* value. By Lemma 6, the correct nodes continue to agree on their *full_clock* value and increase it by "1" at each beat (modulo k).

Therefore, SS-BYZ-CLOCK-SYNC solves the k-Clock problem for any value of k; and converges in expected constant time. □

6. DISCUSSION

6.1 Self-stabilizing Coin-flipping

The main result in this paper is the expected constant time digital clock synchronization algorithm. However, to reach this result an important tool has been developed: the self-stabilizing probabilistic coin-flipping algorithm, which provides a stream of common random bits to all correct nodes. This tool can be useful in developing randomized self-stabilizing solutions to various problems, since it provides a self stabilizing access to a stream of shared coins.

For example, the algorithm in [9] could be adapted to use SS-BYZ-COIN-FLIP as a self-stabilizing coin-flipping building block. Such a change would lead to an exponential reduction in the convergence time of [9]. However, [9]'s convergence time is dependent upon the wraparound clock value, and therefore would still not be constant.

Notice that the random bit produced at beat r is "independent" of events occurring up to beat $r-1$. However, the adversary can "see" the result of the coin at beat r and take it into consideration when sending messages at beat r. Thus, one must be careful when using the stream of random bits; specifically, one must ensure that the states to choose from (using the random bit) have been decided in the previous beat, and not in the current beat. The technique presented in this paper can be adapted in dealing with such situations.

6.2 Self-stabilizing Pipelining

To the best of our knowledge, pipelining as means of transforming non-self-stabilizing *Byzantine* tolerant algorithms into self-stabilizing *Byzantine* tolerant algorithms, was first suggested in [15]. The current work is another example of employing the "pipeline concept" in a self-stabilizing and *Byzantine* tolerant protocol. It is interesting to classify the class of problems that can be solved using this technique.

6.3 Future Research

We consider two main points for future research; the first, regards the bounded-delay model, which assumes there is a bound on messages' delivery time (replacing the global-beat-system assumption). Previously, clock synchronization in the bounded-delay model was solved using an underlaying pulsing algorithm with linear convergence time. Can the ideas in the current paper be transported to the bounded-delay model, and reduce the convergence time to expected constant? This can be done either by directly solving the clock synchronization problem, or by reducing the convergence time of the underlying pulsing algorithm. If so, what extra overhead will be required? Notice that automatic translators from the global-beat-system model to the bounded-delay model exist, but they require linear running time. Therefore, they cannot efficiently transport the current ideas into the bounded-delay model.

The second point regards asynchronous systems. Without probability, it is impossible to solve the clock synchronization problem in an asynchronous network with *Byzantine* nodes. However, once probability is introduced, such a solution might be feasible. It is interesting to see what form the clock synchronization problem takes in an asynchronous setting, and what kind of probabilistic solutions apply.

7. REFERENCES

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

[2] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM.

[3] G. Bracha. An O(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.

[4] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, New York, NY, USA, 1993. ACM.

[5] 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.

[6] D. Dolev and E. N. Hoch. Byzantine self-stabilizing pulse in a bounded-delay model. In *Proc. of 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'07)*, Paris, France, Nov 2007.

[7] D. Dolev and E. N. Hoch. On self-stabilizing synchronous actions despite byzantine attacks. In *Proc. the 21st Int. Symposium on Distributed Computing (DISC'07)*, Lemesos, Cyprus, Sep. 2007.

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

[9] 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.

[10] S. Dolev and J. L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults (abstract). In *PODC*, page 256, 1995.

[11] C. Dwork, D. Shmoys, and L. Stockmeyer. Flipping persuasively in constant time. *SIAM Journal on Computing*, 19(3):472–499, 1990.

[12] P. Feldman and S. Micali. An optimal probabilistic algorithm for synchronous byzantine agreement. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 341–378, London, UK, 1989. Springer-Verlag.

[13] M.J. Fischer and N.A. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters*, 14:183–186, 1982.

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

[15] 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.

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

[17] M. Rabin. Randomized Byzantine generals. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[18] R. Turpin and B. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *INFO. PROC. LETT.*, 18(2):73–76, 1984.