


**Secure and Self-stabilizing Clock Synchronization in Sensor Networks**

Elsevier Editorial System(tm) for Theoretical Computer Science
Manuscript Draft

Manuscript Number:

Title: Secure and Self-Stabilizing Clock Synchronization in Sensor Networks

Article Type: Special Issue: Self-Stabilization (Dolev

Section/Category: A - Algorithms, automata, complexity and games

Keywords: Secure and Resilient Computer Systems, Sensor-Network Systems, Clock-synchronization, Self-Stabilization

Corresponding Author: Dr. Elad Michael Schiller,

Corresponding Author's Institution:

First Author: Jaap-Henk Hoepman

Order of Authors: Jaap-Henk Hoepman; Andreas Larsson; Elad Michael Schiller; Philippas Tsigas

# Secure and Self-Stabilizing Clock Synchronization in Sensor Networks

Jaap-Henk Hoepman, † Andreas Larsson, ‡ Elad M. Schiller, ‡ and Philippas Tsigas ‡

† *TNO ICT, P.O. Box 1416, 9701 BK Groningen, The Netherlands and Radboud University Nijmegen, P.O. BOX 9010, 6500 GL Nijmegen, The Netherlands.* jaap-henk.hoepman@tno.nl

‡ *Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University Rännvägen 6B SE-412 96 Göteborg Sweden Phone/Fax: +46-31-772 1052/+46-31-772 3663.* {larandr,elad,tsigas}@chalmers.se

**Abstract**

In sensor networks, correct clocks have arbitrary starting offsets and nondeterministic fluctuating skews. We consider an adversary that aims at tampering with the clock synchronization by intercepting messages, replaying intercepted messages (after the adversary's choice of delay), and capturing nodes (i.e., revealing their secret keys and impersonating them). We present the first self-stabilizing algorithm for secure clock synchronization in sensor networks that is resilient to such an adversary's attacks. Our algorithm tolerates random media noise, guarantees with high probability efficient communication overheads, and facilitates a variety of masking techniques against pulse-delay attacks in the presence of captured nodes.

*Key words:*
Secure and Resilient Computer Systems, Sensor-Network Systems, Clock-synchronization, Self-Stabilization

## 1. Introduction

Accurate clock synchronization is imperative for many applications in sensor networks, such as mobile object tracking, detection of duplicates, and TDMA radio scheduling. Broadly speaking, existing clock synchronization protocols are too expensive for sensor networks because of the nature of the hardware and the limited resources that sensor nodes have. The unattended environment, in which sensor nodes typically reside, necessitates secure solutions and autonomous system design criteria that are self-defensive against a malicious adversary.

To illustrate an example of clock synchronization importance, consider a mobile object tracking application that monitors objects that pass through the network area (see [2]). Nodes detect the passing objects, record the time of detection, and send the estimated trajectory. Inaccurate clock synchronization would result in an estimated trajectory that could differ significantly from the actual one.

We propose the first self-stabilizing algorithm for clock synchronization in sensor networks with security concerns. We consider an adversary that intercepts messages that it later replays. Our algorithm guarantees automatic recovery after the occurrence of arbitrary failures. Moreover, the algorithm tolerates message omission failures that might occur, say, due to the algorithm's message collisions or due to random media noise.

The short propagation delay of messages in close range wireless communications allows nodes to use broadcast transmissions to approximate pulses that mark the time of real physical events (i.e., beacon messages). In the *pulse-delay* attack, the adversary snoops messages, jams the synchronization pulses, and replays them at the adversary's choice of time (see [9, 10, 20]).

We are interested in fine-grained clock synchronization, where there are no cryptographic counter measures for such pulse-delay attacks. For example, the *nonce* techniques strive to verify the freshness of a message by issuing pseudo-random numbers for ensuring that old communications could not be reused in replay attacks (see [19]). Unfortunately, the lack of fine-grained clock synchronization implies that the round-trip time of message exchange cannot be efficiently estimated. Therefore, it is not clear how the nonce technique can detect pulse-delay attacks.

The system strives to synchronize its clocks while forever monitoring the adversary. We assume that the adversary cannot break existing cryptographic primitives for sensor networks by eavesdropping (e.g., [19, 23]).

However, we assume that the adversary can *capture* nodes, reveal their entire state (including private variables), stop their execution, and impersonate them.

We assume that, at any time, the adversary has a distinct location in space and a bounded influence radius, uses omnidirectional broadcasts from that distinct location, and cannot intercept broadcasts for an arbitrarily long period. (Namely, we consider system settings that are comparable to the settings of Gilbert et al. [11], that consider the minimal requirements for message delivery under broadcast interception attacks.) We explain how, by following these realistic assumptions, we can sift out responses to delayed beacons.

A secure synchronization protocol should mask attacks by an adversary that aims to make the protocol give an erroneous output. Unfortunately, due to the unattended environment and the limited resources, it is unlikely that all the designer's assumptions hold forever, e.g., over time the number of captured nodes becomes sufficiently large for the adversary to tamper with the clock.

We consider systems that have the capability of monitoring the adversary, and then stopping it by external intervention. In this case, the nodes start executing their program from an arbitrary state. From that point on, we require rapid system recovery. Self-stabilizing algorithms [3, 4] cope with the occurrence of transient faults in an elegant way. Self-stabilizing systems can be started in *any* configuration, that might occur due to the occurrence of an arbitrary combination of failures. From that arbitrary starting point, the algorithm must ensure that it accomplishes its task if the system obeys the designer's assumptions for a sufficiently long period.

We focus on the fault-tolerance aspects of secure clock synchronization protocols in sensor networks. Our objective is to design a distributed algorithm for sampling $n$ clocks in the presence of $t$ incorrect nodes (i.e., faulty or captured). The clock sampling algorithm facilitates clock synchronization using a variety of existing masking techniques to overcome pulse-delay attacks in the presence of captured nodes, e.g., [9, 10] use Byzantine agreement (requires $3t+1 \leq n$), and [20] considers the statistical outliers (requires $2t + \epsilon \leq n$, where $\epsilon \in O(1)$).

The execution of a clock synchronization protocol can be classified between two extremes: *on-demand* and *continuous*. Nodes that wish to synchronize their clocks can invoke a distributed procedure for clock synchronization on-demand. The procedure terminates as soon as the nodes reach

their target precision. An execution of a clock synchronization program is classified as continuous if no node ever stops invoking the clock synchronization procedure.

Our generic design facilitates a trade-off between energy conservation (i.e., on-demand operation) and fine-grained clock synchronization (i.e., continuous operation). The trade-off allows budget policies to balance between application requirements and energy constraints. (More details appear in [16] Section 1.3.2.)

### *1.1. Our Contribution*

We present the first design for secure and self-stabilizing clock synchronization in sensor networks that is resilient to an adversary that can capture nodes and launch pulse-delay attacks. Our design tolerates transient failures that may occur due to temporary violation of the designer's assumption, e.g., the adversary captures more than $t$ nodes and then stops. After the system resumes operation according to designer assumption, the algorithm secures with high probability clock precision that is $O((\log n)^2)$ times the optimum, which requires the communication of at least $O(n^2)$ timestamps, where $n$ is the number of sensor nodes. We assume that (before and after the system's recovery) there are message omission failures, say, due to random media noise or the algorithm's message collision. The correct node sends beacons and responds to the other nodes' beacons. We use a randomized strategy for beacon scheduling that guarantees collision avoidance with high probability.

### *1.2. Document structure*

We start by describing the system settings (Section 2) and formally present the algorithm (Section 3). A proof of the algorithm correctness (Section 4) is followed by performance evaluation (Section 5). Then we review the literature and draw our conclusions (Section 6).

## 2. System Settings

We model the system as one that consists of a set of communicating entities, which we call processors (or nodes). We denote the set of processors by $P$, where $|P| \leq N$; $N$ is an upper bound on the number of processors and is known by the processors themselves. In addition, we assume that every processor $p_i \in P$ has a unique identifier, $i$.

*2.1. Time, Clocks, and Their Notation*

We follow settings that are compatible with that of Herman and Zhang [12]. We consider three notations of time: *real time* is the usual physical notion of continuous time, used for definition and analysis only; *native time* is obtained from a native clock, implemented by the operating system from hardware counters; *local time* builds on native time with an additive adjustment factor in an effort to approximate a cluster-wise clock.

We consider applications that require the clock interface to include the *read* operation, which returns a timestamp with $T$ possible states.[1] Let $C_k^i$ and $c_k^i$ denote the value $p_i \in P$ gets from the $k^{th}$ *read* of the native or local clock, respectively. Moreover, let $r_k^i$ denote the real-time instant associated with that $k^{th}$ *read* operation.

Clock counters do not increment at ideal rates, because the hardware oscillators have manufacturing variations and the rates are affected by voltage and temperature. The clock synchronization algorithm adjusts the local clock in order to achieve synchronization, but never adjusts the native clock. We define the native clocks *offset* $\delta_{i,j}(k,q) = C_k^i - C_q^j$, where $\Delta_{i,j}(k,q) = r_k^i - r_q^j = 0$. We assume that, throughout system execution, the native clock offset is arbitrary. Moreover, the *skew* of $p_i$'s clock $\rho_i = \lim_{\Delta_{i,i}(k,q)\to 0} \delta_{i,i}/\Delta_{i,i}(k,q)$ is in $[\rho_{\min}, \rho_{\max}]$, where $\rho_{\min}$ and $\rho_{\max}$ are known constants. Thus, the clock skew is the first derivative of the clock offset value with respect to real time. Because clock skew is generally not constant, higher order derivatives of the clock rate are nonzero. We define the relation between the native time of processor $p_i$ and the real time. We assume that $\rho_{\min} = 1 - \kappa \leq \rho_i \leq 1 + \kappa = \rho_{\max}$, where 1 is the real time unit and $\kappa \geq 0$. The second derivative of the clock's offset is called *drift*. We follow the approach of Herman and Zhang [12] and allow non-zero drift (as long as $\rho_i \in [\rho_{\min}, \rho_{\max}]$).

*2.2. Communications*

Wireless transmissions are subject to collision and noise. The processors communicate among themselves using local broadcast primitives, *LBcast* and *LBrecv*, with a transmission radius of at most $R_{lb}$. We consider the potential of any pair of processors to communicate directly, or to interfere with each other's communications.

[1] In footnote 5 on page 14 we show what the minimal size of $T$ is.

We associate every processor, $p_i$, with a fixed and unknown location in space, $L_i$. We denote the potential set of processors that processor $p_i \in P$ can directly communicate with (with whose communications, processor $p_i$ can interfere) by $G_i \subseteq \{p_j \in P | R_{lb} \geq |L_i - L_j|\}$ (respectively, $\overrightarrow{G_i} \subseteq \{p_j \in P | 2R_{lb} \geq |L_i - L_j|\}$). Throughout the paper, when we talk about processors, say $p_i$ and $p_j$, we assume that $p_j \in G_i$ unless anything else is specifically mentioned. We note that $G_i$ is not something a processor $p_i$ needs to know in advance, but something it discovers as it receives messages from other nodes. We assume that $n \geq |\overrightarrow{G_i}|$ is a known upper bound on any node's degree.

*2.2.1. Communication Operations*

We model the communication channel, $queue_{i,j}$, from processor $p_i$ to processor $p_j \in G_i$ as a FIFO queuing list of the messages that $p_i$ has sent to $p_j$ and $p_j$ is about to receive. When $p_i$ broadcasts message $m$, the operation *LBcast* inserts a copy of $m$ to every $queue_{i,j}$, such that $p_j \in G_i$. Every message $m \in queue_{i,j}$ is associated with a particular time at which $m$ arrives at $p_j$. Once $m$ arrives, $p_j$ executes *LBrecv*. We require that the period between the time at which $m$ enters the communication channel and the time at which $m$ leaves it, is at most a constant, $d$. We assume that $d$ is a known and efficient upper bound on the communication delay between two neighboring processors.

*2.2.2. Accessing the Communication Media*

We assume that processor $p_i$ uses the following optimization, which is part of many existing implementations. Before accessing the communication media, $p_i$ waits for a period $d$ and broadcasts only if there was no message transmitted during that period. Thus, processor $p_i$ does not intercept broadcasts that it started receiving (and did not finish) before time $t - d$, where $t$ is the time of the broadcast by $p_i$.

*2.2.3. Security Primitives*

The existing literature describes many elements of the secure implementation of the broadcast primitives *LBcast* and *LBrecv* using symmetric key encryption and message authentication (e.g., [19, 23]). We assume that neighboring processors store predefined pairwise secret keys. In other words, $p_i, p_j \in P : p_j \in G_i$ store keys $s_{i,j} : s_{i,j} = s_{j,i}$. The adversary cannot efficiently guess $s_{i,j}$. Confidentiality and integrity are guaranteed by encrypting

the messages and adding a message authentication code. We can guarantee messages' freshness by adding a message counter (coupled with the beacon's timestamp) to the message before applying these cryptographic operations, and by letting receivers reject old messages, say, from the clock's previous incarnation. Note that this requires maintaining, for each sender, the index of the last properly received message. As explained above, the freshness criterion is not a suitable alternative to fine-grained clock synchronization in the presence of pulse-delay attacks.

### *2.3. The Interleaving Model*

Every processor, $p_i$, executes a program that is a sequence of *(atomic) steps*. For ease of description, we assume the interleaving model where steps are executed atomically, a single step at any given time. An input event, which can be either the receipt of a message or a timer going off, triggers each step of $p_i$. Only steps that start from a timer going off may include (at most once) an *LBcast* operation. We note that there could be steps that read the clock and decide not to broadcast.

Since no self-stabilizing algorithm terminates (see [4]), the program of a processor consists of a do-forever loop. An iteration is said to be complete if it starts in the loop's first line and ends at the last (regardless of whether it enters conditional branches). A processor executes other parts of the program (and other programs) and activates the loop upon a time-out. We assume that every processor triggers the loop's time-out within every period of $u/2$, where $u > w + d$ is the *(operation) timeslot*, where $w < u/2$ is the time it takes to execute a complete iteration of the do-forever loop. Since processors execute programs other than the clock synchronization, the actual time, $t$, in which the timer goes off is hard to predict. Therefore, for the sake of simplicity, we assume that time $t$ is uniformly distributed.[2]

The *state* $s_i$ of a processor $p_i$ consists of the value of all the variables of the processor (including the set of all incoming communication channels, $\{queue_{j,i} | p_j \in G_i\}$). The execution of a step in the algorithm can change the state of a processor. The term *system configuration* is used for a tuple of the form $(s_1, s_2, \cdots, s_n)$, where each $s_i$ is the state of processor $p_i$ (including messages in transit for $p_i$). We define an *execution* $E = c[0], a[0], c[1], a[1], \ldots$

[2]We note that a simple random scheduler can be used for the case in which time $t$ does not follow a uniform distribution.

as an alternating sequence of system configurations $c[x]$ and steps $a[x]$, such that each configuration $c[x+1]$ (except the initial configuration $c[0]$) is obtained from the preceding configuration $c[x]$ by the execution of the step $a[x]$. We often associate the notation of a step with its executing processor $p_i$ using a subscript, e.g., $a_i$.

*2.3.1. Tracing Timestamps and Communications*

The communication operations that we use, *LBcast* and *LBrecv*, have a time notation that we call *timestamp*. We assume that all timestamps have $T$ possible states. We assume the existence of an efficient algorithm for timestamping the message in transfer (see [23]).

That is, the sent message includes the estimated value of the native clock at sending time. The timestamp of an *LBcast* operation is the native time at which message $m$ is sent. When processor $p_i$ executes the *LBrecv* operation, an event is triggered with the arguments $j$, $t$, and $\langle m \rangle$: $p_j$ is the sending processor of message $\langle m \rangle$, which $p_i$ receives when $p_i$'s native clock is $t$. We note that every step can be associated with at most one communication operation. Therefore it is sufficient to access the native clock counter only once during or at the end of the operation. We denote by $C^i(a_i)$ the native clock value associated with the communication operation in step $a_i$, which processor $p_i$ takes.

*2.3.2. Adversarial Message Omission and Delay*

We assume that at any time, the adversary and all processors have distinct (unknown) locations in space. We assume that there is a single adversary and that its radio transmitter sends omnidirectional broadcasts (using antennas that radiate equally in space). Therefore, the adversary cannot arbitrarily control the distribution in space of the set of recipients for which the beacon's broadcast is delayed or omitted. We assume that it chooses a sphere that divides the set of processors in two: (1) The correct receivers are outside the sphere and receive all beacons on time, and (2) The late receivers are inside the sphere and each beacon can be received on time, after a delay that is greater than a known constant, or not at all.

*2.3.3. Concurrent vs. Independent Broadcasts*

We say that processor $p_i \in P$ performs an *independent broadcast* in a step $a_i \in E$ if there is no processor $p_j \in \overrightarrow{G_i}$ that broadcasts in a step $a_j \in E$, such that either (1) $a_j$ is performed after $a_i$ and before step $a_k^r$ that receives

the message that was sent in $a_i$ (where $p_k \in G_i$), or (2) $a_i$ is performed after $a_j$ and before step $a^r_{k'}$ that receives the message that was sent in $a_j$ (where $p_{k'} \in G_j$). We say that processor $p_i \in P$ performs a *concurrent broadcast* in a step $a_i$ if $a_i$ is dependent (i.e., "not independent"). Concurrent broadcasts can cause message collisions.

*2.3.4. Fair Communications*

The processors reside in the unattended environment and malicious adversarial activity is not the only reason why communication links may fail. Therefore, we consider message omission due to either random media noise or message collisions that the algorithm causes.

Gilbert et al. [11] consider the minimal requirements for message delivery under broadcast interception attacks and assume that the adversary intercepts no more than $\beta$ broadcasts of the algorithm, where $\beta$ is a known constant. We note that the result of Gilbert et al. is applicable in a model in which, in every period, the algorithm is able to broadcast at most $\alpha$ messages and the adversary can intercept at most $\beta$ of the algorithm's messages. Our system settings are comparable to the assumptions made by Gilbert et al. [11] on the ratio of $\beta/\alpha$. The parameter $\xi \geq 1$ denotes the maximal number of repeated transmissions required for at least one successful message transfer, when failed transmissions due to collisions are not counted. Successful message transfer is when the message is received by all other processors. We assume that all processors know $\xi$.

We say that execution $E$ has *fair communications*, if, whenever processor $p_i$ independently broadcasts $\xi$ successive messages (successive in terms of the algorithm's messages sent by $p_i$) in steps $a^{\xi}_i \in E$, every processor receives at least one of these messages. We note that fair communication does not imply reliable communication even for $\xi = 1$, because processors might broadcast concurrently when there is no agreed broadcast schedule or when the clock synchrony is not tight.

*2.3.5. The Environment*

The environment that restricts the adversary's ability to launch message interception attacks guarantees fair communication. The environment can execute the operation $\mathsf{omission}_i(m)$ (which is associated with a particular message, $m$, sent by processor $p_i$) immediately after $\mathsf{LBcast}_i(m)$. The environment selects a subset of $p_i$'s neighbors ($R_i \subseteq G_i$) to remove any message

$m_i$ from their queues $queue_{i,j}$ (such that $p_j \in R_i$). We assume that the environment arbitrarily selects $R_i$ when invoking *omission* due to the algorithm's message collision. The adversary, under the environment's supervision, selects messages to remove due to random media noise. The adversary launches message interception attacks by selecting $R_i$. The environment supervises so the adversary does not violate the fair communication requirements.

### *2.4. System Specifications*

#### *2.4.1. Fair Executions*

An execution $E$ is *fair* if the communications are fair and every correct processor, $p_i$, executes steps in a timely manner (by letting the loop's timer go off in the manner that we explain above).

#### *2.4.2. The Task*

We define the system's task by a set of executions called *legal executions* ($LE$) in which the task's requirements hold. A configuration $c$ is a *safe configuration* for an algorithm and the task of $LE$ provided that any execution that starts in $c$ is a legal execution (belongs to $LE$). An algorithm is *self-stabilizing* with relation to the task of $LE$ if every infinite execution of the algorithm reaches a safe configuration with relation to the algorithm and the task.

#### *2.4.3. Clock Synchronization Requirements*

Roughly speaking, without any attacks or failures, the native clocks follow similar characteristics. Processors can synchronize their local clocks by revealing these characteristics. The task's output decodes the coefficient vector of a finite degree polynomial $P_{i,j}(t)$ that closely approximates the native clock value of processor $p_j$ at time $t$, where $t$ is a value of $p_i$'s native clock. Römer et al. [17] explain how to calculate $\{P_{i,j}(t)\}_{j \neq i}$.

Elson et al. [7, 6] explain how to calculate the global and the local clocks using $\{P_{i,j}(t)\}_{j \neq i}$. We note that the local $c^i$ could be agreed upon in different manners, one of which is based on clustered networks. In each cluster, every processor considers a predefined set of processors, call the *cluster head*, for which it tries to estimate a common local time using a predefined deterministic function.

This paper presents an algorithm for sampling $n$ neighboring clocks. We measure the algorithm's performance by looking at a bound on the length

of periods in which the random schedule is "nice" (useful). In a nice broadcasting schedule, each of the $n$ processors sends at least one beacon that all other $n-1$ processors respond to with high probability (see Definition 1 and Theorem 1 in Section 4 and Section 5).

Let $p_i$, $p_j$, and $p_k$ be three correct nodes. Suppose that $p_j$ broadcasts a message that $p_k$ receives (after a delay) and $p_k$ then sends a response message that $p_i$ receives (possibly $i = j$). We require that $p_i$ detects that $p_k$ has responded to a delayed message in the presence of at most $t$ captured nodes.

## 3. Secure and Self-Stabilizing Clock Synchronization

In order to explain better the scope of the algorithm, we present a generic organization of secure clock synchronization protocols. The objectives of the clock synchronization protocol are to: (1) periodically broadcast beacons, (2) respond to beacons, and (3) aggregate beacons with their responses in records and deliver them to the upper layer. Every node estimates the local clock after sifting out responses to delayed beacons. Unlike objectives (1) to (3), the clock estimation task is not a hard realtime task. Therefore, the algorithm outputs records to the upper layer that synchronizes the local clock after neutralizing the effect of pulse-delay attacks (see section 6 for more details). The algorithm focuses on the following two tasks:

• *Beacon Scheduling:* The nodes sample clock values by broadcasting beacons and waiting for their response. The task is to guarantee round-trip message exchange.

• *Beacon and Response Aggregation:* Once a beacon completes the round-trip exchange, the nodes deliver to the upper layer the records of a beacon and its set of responses.

We present a design for an algorithm that samples clocks of neighboring processors by continuously sending beacons and responses. Without synchronized clocks, the nodes cannot efficiently follow a predefined schedule. Moreover, assuring reliable communication becomes hard in the presence of random media noise and message collision. The celebrated Aloha protocol [1] (which does not consider nondeterministic fluctuating skews) inspires us to take a randomized strategy for scheduling broadcasts and overcome the above difficulties by showing that with high probability the neighboring processors are able to exchange beacons and responses within a short period. Our scheduling strategy is simple; the processors choose a random time to

broadcast from a predefined period $D$. We use a redundant number of broadcasting timeslots in order to overcome the clocks' asynchrony. Moreover, we use a parameter, $\ell$, used to trade off between the minimal size of $D$ and the probability of having a collision-free schedule.

*3.1. Beacon and Response Aggregation*

The algorithm allows the use of clock synchronization techniques such as *round-trip synchronization* [9, 10] and *reference broadcasting* [6]. For example, in the round trip synchronization technique, the sender $p_j$ sends a timestamped message $\langle t_1 \rangle$ to receivers, $p_k$, which receive the message at time $t_2$. The receiver $p_k$ responds with the message $\langle t_1, t_2, t_3 \rangle$, which $p_k$ sends at time $t_3$ and $p_j$ receives at time $t_4$. Thus, the output records are in the form of $\langle j, t_1, \{\langle k, \langle t_2, t_3, t_4 \rangle\rangle\}\rangle$, where $\{\langle k, \langle t_2, t_3, t_4 \rangle\rangle\}$ is the set of all received responses sent by nodes $p_k$.

We piggyback beacon and response messages. For the sake of presentation simplicity, let us start by assuming that all beacon schedules are in a (deterministic) Round Robin fashion. Given a particular node $p_i$ and a particular beacon that $p_i$ sends at time $t_s^i$, we define $t_s^i$'s *round* as the set of responses, $\langle t_s^j, t_r^j \rangle$, that $p_i$ sends to node $p_j$ for $p_j$'s previous beacon, $t_s^j$, where $t_r^j$ is the time in which $p_i$ received $p_i$'s beacon $t_s^j$. Node $p_i$ piggybacks its beacon with the responses to nodes, $p_j$, and the beacon message, $\langle v_i \rangle$, is of the form: $\langle \langle t_s^1, t_r^1 \rangle, \ldots \langle t_s^{i-1}, t_r^{i-1} \rangle, t_s^i, \langle t_s^{i+1}, t_r^{i+1} \rangle, \ldots \langle t_s^n, t_r^n \rangle \rangle$.

Now, suppose that the schedules are not done in a Round Robin fashion. We denote $p_j$'s sequence of up to $BLog$ most recently sent beacons with $[t_s^j(k)]_{0 \leq k < BLog}$, among which $t_s^j(k)$ is the *k-th* oldest and $BLog$ is a predefined constant.[3] We assume that, in every schedule, $p_i$ receives at least one beacon from $p_j$ before broadcasting $BLog$ beacons. Therefore, $p_i$'s beacon message, $\langle v_i \rangle$, can include a response to $p_j$'s most recently received beacon, $t_s^j(k)$, where $0 \leq k < BLog$.

Since not every round includes a response to the last beacon that $p_i$ sends, then $p_i$ stores its last $BLog$ beacon messages in a FIFO queue, $q_i[k] = [t_s^j]_{0 \leq k < BLog}$. Moreover, every beacon message includes all responses to the $BLog$ most recently received beacons from all nodes. Let $q_j = q[k]_{0 \leq k < BLog}$ be $p_i$'s FIFO queue of the last $BLog$ records of the form

[3]We note that $BLog$ depends on the safety parameter, $\ell$, for assuring that all nodes successfully broadcast and other parameters such as the number of processors, $n$, and the bound on clock skews $\rho_{\min}$ and $\rho_{\max}$ (see Section 2).

$\langle t_s^j(k), t_r^j(k)\rangle$, among which $t_s^j(k)$ is $p_i$'s *k-th* oldest beacon from $p_j$, $t_r^j(k)$ is the time at which it was received and $i \neq j$. The new form of the beacon message is: $\langle q_1, \ldots q_{i-1}, q_i, q_{i+1}, \ldots q_n\rangle$. In the round trip synchronization, the nodes take the role of a *synchronizer* that sends the beacon and waits for responses from the other nodes. The program of node $p_i$ considers both cases in which $p_i$ is, and is not, respectively, the synchronizer.

*3.2. The Algorithm's Pseudo-code*

The pseudo-code, in Figure 2, includes two procedures: (1) a do-forever loop that schedules and broadcasts beacon messages (lines 58 to 70) and (2) an upon message arrival procedure (lines 72 to 76).

*3.2.1. The Do-Forever Loop*

The do-forever loop periodically tests whether the "timer" has expired (in lines 59 to 64).[4] In case the beacon's next schedule is "too far in the past" or "too far in the future", then processor $p_i$ "forces" the "timer" to expire (line 61). The algorithm tests that all the stored beacon messages are ordered correctly and refer to the last $BLog$ beacons (line 62). In the case where the recorded information about beacon messages is incorrect, the algorithm flushes the queues (line 63).

When the timeslot arrives the processor outputs a synchronizer case record for the beacon that processor $p_i$ has sent $BLog$ rounds ago (line 66). It contains for each of the other processors, $p_j$, the receive time of that beacon. Moreover, it contains for processor $p_j$, the send and receive times for a message back from $p_j$ to $p_i$. These data can be used for the round-trip synchronization and delay detection in the upper layer. Then, $p_i$ enqueues the timestamp of the beacon it is about to send during this schedule (line 67). The next schedule for processor $p_i$ is set (lines 68 and 69) just before it broadcasts the beacon message (line 70).

*3.2.2. The Message Arrival*

When a beacon message arrives (line 72), processor $p_i$ outputs a record of the non-synchronizer case (line 74). These data can be used for the reference

[4]Recall that by our assumptions on the system settings (Section 2), the do-forever loop's timer will go off within any period of $u/2$. Moreover, since the actual time cannot be predicted, we assume that the actual schedule has a uniform distribution over the period $u$. (A straightforward random scheduler can assist, if needed, to enforce the last assumption.)

broadcast in the upper layer. Once $p_i$ receives a beacon from processor $p_j$, $p_i$ scans $m[\ ]$ for responses that refer to the oldest beacon from $p_j$ that $p_i$ have received and messages back, from receiving nodes $p_k$, to $p_j$. Now that the information connected to the oldest beacon from $p_j$ has been output, processor $p_i$ can store the arrival time of newly received message (line 75) and the message itself (line 76).

## 4. Correctness

In this section we demonstrate that the task of random broadcast scheduling is achieved by the algorithm that is presented in Figure 2. Namely, with high probability, the scheduler allows the exchange of beacons and responses within a short time. The objectives of the task of random broadcast scheduling are defined in Definition 1 and consider *broadcasting rounds*.

**Definition 1 (Nice executions).** *Let us consider the executions of the algorithm presented in Figure 2. Let $\Gamma(n)$ be the set of all execution prefixes, $E_{\Gamma(n)}$, such that within the first $\mathcal{R}$ broadcasting rounds of $E_{\Gamma(n)}$, each of the $n$ processors send at least one beacon that all other $n-1$ processors respond to. We say that execution $E$ is* nice *if, with high probability, $E$ has a prefix in $\Gamma(n)$.*

The proof of Theorem 1 (Section 4.3, page 26) demonstrates that when considering $\mathcal{R} = 2R$, the algorithm reaches nice execution with probability of at least $1 - 2^{-\ell+1}$, where $\ell$ is a predefined constant and $R = \lceil \xi \frac{\ell + \log_2((\lceil \rho_{\max}/\rho_{\min} \rceil + 1)n)}{-\log_2(1-1/e)} \rceil$ is the expected time it takes all processors to each broadcast at least one message that is received by all other processors.[5]

Once the system reaches a nice execution and the exchange of beacons and responses occurs, the following holds. For every processor $p_i \in P$, there is a set, $S_i$, of beacon records that are in the queues of $m_i$ and the records that were delivered to the upper layer. The set $S_i$ includes a subset, $S'_i \subseteq S_i$, of records for beacons that were sent during the last $\mathcal{R}$ (Definition 1) broadcasting rounds. In $S'_i$, it holds that every processor $p_j \in G_i - \{i\}$ has a beacon record, $rec_j$, such that every processor $p_k \in G_i \cap G_j - \{j\}$ has a

[5] To distinguish between timestamps that should be regarded as being in the past and timestamps that should be regarded as being in the future, we require that $T > 4\mathcal{R}$. In other words, we want to be able to consider at least 2 round-trips in the past and 2 round-trips in the future.

**Constants**:

$i$ = *id of executing processor*
$n$ = *total number of processors*
$w$ = *compensation time between lines* 59 *and* 70
$d$ = *upper bound on message propagation delay*
$u$ = *size of a timeslot in time units* ($u > d + w$)
$\ell$ = *a tuning parameter to the trade off between the*
$BLog = 2\lceil \xi \frac{\ell + \log_2((\lceil \rho_{\max}/\rho_{\min} \rceil + 1)n)}{-\log_2(1-1/e)} \rceil$, *backlog size*
$D = 3\ n\ (\lceil \rho_{max}/\rho_{min} \rceil + 1)$, *the broadcast timeslots*
$T$ = *number of possible states of a timestamp*
$\rho_{\min}$ = *lower bound on clock skew*
$\rho_{\max}$ = *upper bound on clock skew*

**Variables**:

$m[n]$ = *all received messages and timestamp*
  *each entry is an array* $v[n]$
    *each entry is a queue* $q[BLog]$
      *each entry is a pair* $\langle s, r \rangle$
*native_clock* : *immutable storage of the native clock*
*cslot* : [0, *D*-1 ] = *current timeslot in use*
*next* : [0, *T* -1 ] = *schedule of next broadcast*
$cT$ = *last do-forever loop′s timestamp*

**External functions**:

output($R$) : *delivers record R to the upper layer*
choose($S$) : *uniform selection of an item from the set S*
enqueue($Q$) : *adds an element to the end of the queue Q*
dequeue($Q$) : *removes the front element of the queue Q*
size($Q$) : *size of the queue Q*
size($Q$) : *size of the queue Q*
first($Q$) : *least recently enqueued element in Q, number* 0
last($Q$) : *most recently enqueued element in Q*
full($Q$) : *whether queue Q is full*
flush($Q$) : *empties the queue Q*
get_s($Q$) : *list elements of field s in Q*
get_r($Q$) : *list elements of field r in Q*

**Macros and inlines**:

border($t$) : $(D\text{-}cslot)u + t \mod T$
schedule($t$) : $cslot \cdot u + t \mod T$
leq($x, y$) : $(\exists\, b : 0 \le b \le 2\ BLog\ D\ u \wedge y \mod T = x + b \mod T)$
enq($q, m$) : {**while** full($q$) **do dequeue**($q$); **enqueue**($m$) }

checklist($q$,$t$) : (∗ *Checks that all elements of a list are chronologically ordered and not in the future* ∗)
  size($q$) = 0 ∨ (leq(first($q$),$t$) ∧ leq(last($q$),$t$) ∧ {∀ $b_1 < b_2$, {$b_1$,$b_2$} ⊆ [1,size($q$) ] : leq($q[b_1]$,$q[b_2]$)}
check_s($t$) : checklist(get_s($m[i].v[i].q$),$t$) (∗ *Coherency test for the send times of a processor′s own beacons* ∗)
check_r($t$) : ∧ {∀ $j \in P$-{$i$} : checklist(get_r($m[i].v[j].q$),$t$)} (∗ *Coherency test for the received beacon times* ∗)

(∗ *Get response-record for* $p_k$, *for* $p_j$ *as the synchronizer* ∗)
ts($s, j, k$) : {**if** ($\exists\ b_1^j, b_2^j, b_1^k, b_2^k$:
  $s = m[j].v[j].q[b_1^j].s = m[k].v[j].q[b_1^k].s \wedge$
  $m[k].v[k].q[b_2^k].s = m[j].v[k].q[b_2^j].s \wedge$
  leq($m[j].v[j].q[b_1^j].s$, $m[j].v[k].q[b_2^j].r$) ∧
  leq($m[k].v[j].q[b_1^k].r$, $m[k].v[k].q[b_2^k].s$)) **then**
  **return** $\langle m[k].v[j].q[b_1^k].r$, $m[k].v[k].q[b_2^k].s$, $m[j].v[k].q[b_2^j].r \rangle$
  **else return** ⊥}

Figure 1: Constants, variables, external functions, and macros for the secure and self-stabilizing native clock sampling algorithm in Figure 2.

beacon record, $rec_k$, that includes a response to $rec_j$. In other words, $\mathcal{R}$ is a bound on the length of periods for which processor $p_i$ needs to store beacon records. Moreover, with high probability, within $\mathcal{R}$ broadcasting rounds, $p_i$ gathers $n$ beacons and their responses from all other $n-1$ processors. For this reason, we set $BLog$ to be $\mathcal{R}$.

```
58 Do forever, every u/2
     let cT = read(native_clock) + w
60   if ¬ (leq(next-2Du, cT) ∧ leq(cT, next+u)) then
       next ← cT
62   if ¬ (check_s(cT) ∧ check_r(cT)) then
       ∀ j,k ∈ P : flush(m[j].v[k].q)
64   if leq(next, cT) ∧ leq(cT, next + u) then
       let s = first(m[i].v[i].q).s
66     output ⟨i, s, { ⟨j, ts(s, i, j)⟩ : j ∈ G_i -{i}} ⟩
       enq(m[i].v[i].q, ⟨cT, ⊥⟩)
68     (next, cslot) ← (border(next), choose([0, D-1]))
       next ← schedule(next)
70     LBcast(m[i])

72 Upon LBrecv(j, r, v)          (* i ≠ j *)
     let s = first(m[i].v[j].q).s
74   output ⟨j, s, {⟨k, ts(s, j, k)⟩ : k ∈ G_i -{j}}⟩
     enq(m[i].v[j].q, ⟨last(v[j].q).s, r⟩)
76   m[j] ← v
```

Figure 2: Secure and self-stabilizing native clock sampling algorithm (code for $p_i \in P$).

### *4.1. Scenarios in which balls are thrown into bins*

We simplify the presentation of the analysis by depicting different system settings in which the message transmissions are described by a set of scenarios in which balls are thrown into bins. The sending of a message by processor $p_i$ corresponds to a player $\hat{p}_i$ throwing a ball. Time is discretized into timeslots that are long enough for a message to be sent and received within. The timeslots are represented by an unbounded sequence of bins, $[b_k]_{k\in\mathbb{N}}$. Transmitting a message during a timeslot corresponds to throwing a ball towards and *aiming a ball at* the corresponding bin.

Before analyzing the general system settings, we demonstrate simpler settings to acquaint the reader with the problem. Concretely, we look at the settings in which the clocks of the processors are perfectly synchronized and the communication channels have no noise. We ask the following question: How many bins are needed for every player to get at least one ball, that is not lost due to collisions, in a bin (Lemma 1 and 2)? We then relax the assumptions on the system settings by considering different clock offsets (Claim 2) and by considering different clock skews (Claim 3). We continue by considering noisy communication channels (Claim 4) and conclude the analysis by considering general system settings (Corollary 1).

#### *4.1.1. Collisions*

A message collision corresponds to two or more balls aimed at the same bin. We take the pessimistic assumption that when balls are aimed at neighboring bins, they collide as well. This is to take non-discrete time (and later on, different clock offsets) into account. Broadcasts that "cross the borders"

between timeslots are assumed to collide with messages that are broadcasted in either bordering timeslot. Therefore, in the scenario in which balls are thrown into bins, two or more balls aimed at the same bin or bordering bins will bounce out, i.e., not end up in the bin.

**Definition 2.** *When aiming balls at bins in a sequence of bins, a* successful ball *is a ball that is aimed at a bin b. Moreover, it is required that no other ball is aimed at b or a neighboring bin of b. A* neighboring bin *of b is the bin directly before or directly after b. An* unsuccessful ball *is a ball that is not successful.*

*4.1.2. Synchronous timeslots and communication channels that have no noise*

We prove a claim that is needed for the proof of Lemma 1.

**Claim 1.** *For all $n \geq 2$ it holds that*

$$\left(1 - \frac{1}{n}\right)^{n-1} > \frac{1}{e}. \tag{1}$$

**Proof:** It is well known that

$$\left(1 + \frac{1}{n}\right)^{n} < e \tag{2}$$

for any $n \geq 1$. From this follow that

$$\begin{aligned} \left(1 - \frac{1}{n}\right)^{n-1} &= \left(\frac{n-1}{n}\right)^{n-1} = \left(\frac{n}{n-1}\right)^{-(n-1)} \\ &= \left(1 + \frac{1}{n-1}\right)^{-(n-1)} = \frac{1}{\left(1 + \frac{1}{n-1}\right)^{n-1}} > \frac{1}{e} \end{aligned} \tag{3}$$

for $n \geq 2$. ■

Lemmas 1 and 2 consider an unbounded sequence of bins that are divided into "circular" subsequences that we call *partitions*. We simplify the presentation of the analysis by assuming that the partitions are independent. Namely, a ball that is aimed at the last bin of one partition normally counts as a collision with a ball in the first bin of the next partition. With this assumption, a ball aimed at the last bin and a ball aimed at the first bin in

the same partition count as a collision instead. These assumptions do not cause a loss of generality, because the probability for balls to collide does not change. It does not change because the probability for having a certain number of balls in a bin is symmetric for all bins.

We continue by proving properties of scenarios in which balls are thrown into bins. Lemma 1 states the probability of a single ball being unsuccessful.

**Lemma 1.** *Let $n$ balls be, independently and uniformly at random, aimed at partitions of $3n$ bins. For a specific ball, the probability that it is not successful is smaller than $1 - 1/e$.*

**Proof:** Let $b$ be the bin that the specific ball is aimed at. For the ball to be successful, there are 3 out of the $3n$ bins that no other ball should be aimed at, $b$ and the two neighboring bins of $b$. The probability that no other (specific) ball is aimed at any of these three bins is

$$1 - \frac{3}{3n}. \tag{4}$$

The different balls are aimed independently, so the probability that none of the other $n - 1$ balls are aimed at bin $b$ or a neighboring bin of $b$ is

$$\left(1 - \frac{3}{3n}\right)^{n-1} = \left(1 - \frac{1}{n}\right)^{n-1}. \tag{5}$$

With the help of Claim 1, the probability that at least one other ball is aimed at $b$ or a neighboring bin of $b$ is

$$1 - \left(1 - \frac{1}{n}\right)^{n-1} < 1 - \frac{1}{e}. \tag{6}$$

■

Lemma 2 states the probability of any player not having any successful balls after a number of throws.

**Lemma 2.** *Consider $R$ independent partitions of $D = 3n$ bins. For each partition, let $n$ players aim one ball each, uniformly and at random, at one of the bins in the partition. Let $R \geq (\ell + \log_2 n)/(-\log_2 p)$, where $p = 1 - 1/e$ is an upper bound on the probability of a specific ball to be unsuccessful in a partition. The probability that any player gets no successful ball is smaller than $2^{-\ell}$.*

**Proof:** By Lemma 1 the probability that a specific ball is unsuccessful is upper bounded by $p = 1-1/e$. The probability that a player does not get any successful ball in any of $R$ independent partitions is therefore upper bounded by $p^R$.

Let $X_i$, $i \in [1, n]$ be Bernoulli random variables with the probability of a ball to be successful that is upper bounded by $p^R$:

$$X_i = \begin{cases} 1 & \text{if player } i \text{ gets no successful ball in } R \text{ partitions} \\ 0 & \text{if player } i \text{ gets at least one successful ball in } R \text{ partitions} \end{cases} \tag{7}$$

Let $X$ be the number of players that gets no successful ball in $R$ partitions:

$$X = \sum_{i=1}^{n} X_i \tag{8}$$

The different $X_i$ are a finite collection of discrete random variables with finite expectations. Therefore we can use the Theorem of Linearity of Expectations [15]:

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] \leq \sum_{i=1}^{n} p^R = np^R \tag{9}$$

The random variables assumes only non-negative values. Markov's Inequality [15], $\Pr(X \geq a) \leq \mathrm{E}[X]/a$, therefore gives us:

$$\Pr(X \neq 0) = \Pr(X \geq 1) \leq \frac{\mathrm{E}[X]}{1} \leq np^R \tag{10}$$

For $np^R \leq 2^{-\ell}$ we get that $\Pr(X \neq 0) \leq 2^{-\ell}$, which gives us

$$\begin{aligned} np^R &\leq 2^{-\ell} \Rightarrow \\ \log_2(np^R) &\leq -\ell \Rightarrow \\ \log_2(n) + R\log_2(p) &\leq -\ell \Rightarrow \\ R &\geq \frac{-\ell - \log_2 n}{\log_2 p} = \frac{\ell + \log_2 n}{-\log_2 p}. \end{aligned} \tag{11}$$

■

We now turn to relaxing the simplifying assumptions of synchronized clocks and communication channels with no noise. We start by considering clock offsets and skews. We then consider noisy communication channels.

*4.1.3. Clock offsets*

The clocks of the processors have different offsets and therefore the timeslot boundaries are not aligned. We consider a scenario that is comparable to system settings in which clocks have offsets. In the scenario of balls that are thrown into bins, offsets are depicted as throwing a ball that hits the boundary between bins and perhaps hitting the boundary between partitions.

Claim 2 considers players that have individual sequences of bins. Each sequence has its own alignment of the bin boundaries. Namely, a bin of one player may "overlap" with more than one bin of another player. Thus, the different bin sequences that have different alignments correspond to system settings in which clocks have different offsets.

The proof of Claim 2 describes a variation of the scenario in which balls are thrown into bins. In the new variation, balls aimed at overlapping bins will bounce out. For example, consider two balls aimed at bin $b_i^k$ and $b_j^{k'}$ respectively. If bins $b_i^k$ and $b_j^{k'}$ overlap the balls will cause each other to bounce out.

**Claim 2.** *Consider the scenario in which balls might hit the bin boundaries and take $R$ and $D$ as defined in Lemma 2. Then, we have that the probability that any player gets no successful ball is smaller than $2^{-\ell}$.*

**Proof:** The proof is demonstrated by the following two arguments.

**Hitting the boundaries between bins** From the point of view of processor $p_i$, a timeslot might be the time interval $[t, t+u)$, whereas for processor $p_j$ the timeslot interval might be different and partly belong to two different timeslots of $p_i$. When considering the scenario in which balls are thrown into bins, we note that a bin of one player might be seen as parts of two bins of another player.

In other words, every player, $\hat{p_i}$, has its own view, $[b_k^i]_{k\in\mathbb{N}}$, of the bin sequence $[b_k]_{k\in\mathbb{N}}$. The sequence $[b_k]_{k\in\mathbb{N}}$ corresponds to an ideal discretization of the real time into timeslots, whereas the sequence $[b_k^i]_{k\in\mathbb{N}}$, corresponds to a discretization of processor $p_i$'s native time into timeslots. We say that the bins $[b_k^i]$ and $[b_{k'}^j]$ overlap when the corresponding real time periods of $[b_k^i]$ and $[b_{k'}^j]$ overlap.

Lemma 2 regards balls aimed at neighboring bins as collisions. We recall the requirements that are made for ball collisions (see Section 4.1.1). These requirements say that balls aimed at neighboring bins in $[b_k]_{k\in\mathbb{N}}$, will bounce out. The proof is completed by relaxing the requirements that are made

for ball collisions in $[b_k]_{k\in\mathbb{N}}$. Let us consider the scenario in which players $\hat{p_i}$ and $\hat{p_j}$ aim their balls at bins $b^i_k$ and $b^j_{k'}$ respectively, such that both $b^i_k$ and $b^j_{k'}$ overlap. The bin $b^i_k$ can either overlap with the bins $b^j_{k'-1}$ and $b^j_{k'}$ or (exclusively) overlap with the bins $b^j_{k'}$ and $b^j_{k'+1}$. Balls aimed at any of the bins possibly overlapping with $b^i_k$ (namely $b^j_{k'-1}$, $b^j_{k'}$ and $b^j_{k'+1}$) are regarded as colliding with the ball of player $\hat{p_j}$. The same argument applies to bin $b^j_{k'}$ overlapping with bins $b^i_{k-1}$, $b^i_k$ and $b^i_{k+1}$. In other words, the scenario of Lemma 2, without offset and neighboring bins leading to collision, is a superset in terms of bin overlap to the scenario in which offsets are introduced.

**Hitting the boundaries between partitions** Even if timeslot boundaries are synchronized, processor $p_i$ might regard the time interval $[t, t+Du)$ as a partition, whereas processor $p_j$ might regard the interval $[t, t+Du)$ as partly belonging to two different partitions. When considering the scenario in which balls are thrown into bins, it means that the players' view on which bins are part of a partition can differ.

For each bin, the probability that a specific player chooses to aim a ball at that bin is $1/D$, where $D$ is the number of bins in the partition. Therefore the probability for a ball to be successful does not depend on how other players partition the bins. ■

*4.1.4. Clock skews*

The clocks of the processors have different skews. Therefore, we consider a scenario that is comparable to system settings in which clocks have skews.

In Claim 3, we consider players that have individual sequences of bins. Each sequence has its own bin size. The size of player $\hat{p_i}$'s bins is inversely proportional to processor $p_i$'s clock skew, say $1/\rho_i$. We assume that the balls that are thrown by any player can fit into the bins of any other player. (Say the ball size is less than $1/\rho_{\max}$.) Thus, the different bin sizes correspond to system settings in which clocks have different skews.

Let us consider the number of balls that player $\hat{p_i}$ may aim at bins that overlap with bins in a partition of another player. Suppose that player $\hat{p_i}$ has bins of size $1/\rho_{\max}$ and that player $\hat{p_j}$ has bins of size $1/\rho_{\min}$. Then player $\hat{p_i}$ may aim up to $\hat{\rho} = \lceil \rho_{\max}/\rho_{\min} \rceil + 1$ balls in one partition of player $\hat{p_j}$.

**Claim 3.** *Consider the scenario with clock skews and take $R$ and $D$ as defined in Lemma 2. Let $p = 1 - 1/e$ be an upper bound on the probability of a specific ball to be unsuccessful in a partition. By taking $R_{skew} = R \geq$*

$(\ell + \log_2 \hat{\rho}n))/(-\log_2 p) \in O(\ell + log(n))$, *we have that the probability that any player gets no successful ball is smaller than* $2^{-\ell}$.

**Proof:** By taking the pessimistic assumption that all players see the others, as well as themselves, as throwing $\hat{\rho}$ balls each in every partition we have an upper bound on how many balls can interfere with each other in a partition. Thus by taking partitions of $D = 3\hat{\rho}n$ bins instead of the $3n$ bins of Lemma 2, and substituting $n$ for $\hat{\rho}n$ in the $R$ of Lemma 2,

$$R \geq \frac{\ell + \log_2 \hat{\rho}n)}{-\log_2 p} \in O(\ell + log(n)), \tag{12}$$

the guarantees of Lemma 2 hold. ■

*4.1.5. Communication channels with noise*

In our system settings, message loss occurs due to noise and not only due to the algorithm's message collisions. Recall that $\xi$ defines the number of broadcasts required in order to guarantee at least one broadcast that is not lost due to noise in the channel (see Section 2.3.4). In the scenario in which balls are thrown into bins, this correspondingly means that at most $\xi - 1$ balls are lost to the player's trembling hand for any of its $\xi$ consecutive throws.

**Claim 4.** *Consider the communication channels with noise and take* $R$ *and* $D$ *as defined in Lemma 2. By taking* $R_{noise} \geq \xi R$, *we have that the probability that any player gets no successful ball is smaller than* $2^{-\ell}$.

**Proof:** By the system settings (Section 2), the noise in the communication channels is independent of collisions. We take the pessimistic approach and assume that when a ball is lost to noise, it can still cause other balls to be unsuccessful (just as if it was not lost to noise).

In order to fulfill the requirements of Lemma 2, we can take $\xi R$ partitions instead of $R$ partitions. This will guarantee that each player gets at least $R$ balls that are not lost due to noise and will not change the asymptotic number of bins. ■

*4.1.6. General system settings*

The results gained from studying the scenario in which balls are thrown into bins are concluded by Corollary 1, which is demonstrated by Lemma 2 and claims 2, 3, and 4.

**Corollary 1.** *Suppose that every processor broadcast once in every partition of $D$ timeslots. The probability that all processors successfully broadcast at least one beacon within $R$ partitions is at least $1 - 2^{-\ell}$, when*

$$D = 3\hat{\rho}n \in O(n) \tag{13}$$

$$R = \lceil \xi \frac{\ell + \log_2(\hat{\rho}n)}{-\log_2 p} \rceil \in O(\ell + \log n) \tag{14}$$

$$\hat{\rho} = \lceil \rho_{max}/\rho_{min} \rceil + 1. \tag{15}$$

Corollary 1 shows that within a logarithmic number of broadcasting rounds, all processors can successfully broadcast.

*4.2. The task of random broadcast scheduling*

So far, we have analyzed a general scenario in which balls are thrown into bins. We now turn to show that the scenario indeed depicts the implementation of the algorithm (that is presented in Figure 2).

Hereafter, when we talk about the execution of, or complete iteration of, lines 59 to 70, we do not imply that the branch in lines 65 to 70 are executed in that step (although that can be the case).

**Definition 3 (Safe configurations).** *Let $E$ be a fair execution of the algorithm presented in Figure 2 and $c \in E$ a configuration in which $\alpha_i = (\mathit{leq}(next_i - 2Du, cT_i) \wedge \mathit{leq}(cT_i, next_i)$ holds for every processor $p_i$. We say that $c$ is safe with respect to $LE$.*

We show that $cT_i$ follows the native clock. Namely, the value of $cT_i - w$ is in $[C^i - u, C^i]$.

**Lemma 3.** *Let $E$ be a fair execution of the algorithm presented in Figure 2, and $c$ a configuration that is at least $u$ after the starting configuration. Then, it holds that $(\mathit{leq}(C^i - u, cT_i - w) \wedge \mathit{leq}(cT_i - w, C^i))$ in $c$.*

**Proof:** Since $E$ is fair, the do-forever loop's timer goes off in every period of $u/2$. Hence, within a period of $u$, processor $p_i$ performs a complete iteration of the do-forever loop in an atomic step $a_i$.

Suppose that $c$ immediately follows $a_i$. According to line 59, the value of $cT_i - w$ is the value of $C^i$ in $c$. Let $t = cT_i - w = C^i$. It is easy to see that $\mathit{leq}(t - u, t) \wedge \mathit{leq}(t, t)$ in $c$.

Let $a_i^r$ be an atomic step that includes the execution of lines 73 to 76, follows $c$, and immediately precedes $c' \in E$. Let $t' = C^i$ in $c'$. Then, within a period of at most $u/2$, processor $p_i$ executes step $a_i' \in E$, which includes a complete iteration of the do-forever loop. Since the period between $a_i$ and $a_i'$ is at most $u/2$, we have that $t' - t < u/2$. Therefore $\mathit{leq}(C^i - u, cT_i - w)$ holds in $c'$ as $\mathit{leq}(C^i, cT_i - w)$ holds in $c$. It also follows that $\mathit{leq}(cT_i - w, C^i))$ holds in $c'$ as $C^i = cT_i - w$ in $c$. ■

We show that when a processor $p_i$ executes lines 65 to 70 of the algorithm presented in Figure 2 it reaches a configuration in which $\alpha_i$ holds. This claim is used in Lemma 4 and Lemma 5.

**Claim 5.** *Let $E$ be a fair execution of the algorithm presented in Figure 2. Moreover, let $a_i \in E$ a step that includes a complete iteration of lines 59 to 70 and $c$ the configuration that immediately follows $a_i$. Suppose that processor $p_i$ executes lines 65 to 70 in $a_i$, then $\alpha_i$ holds in $c$.*

**Proof:** Among the lines 65 to 70, only lines 68 to 69 can change the values of $\alpha_i$. Let $t_1 = next_i$ immediately after line 64 and let $t_2 = next_i$ immediately after the execution of line 69. We denote by $A = t_2 - t_1$ the value that lines 68 to 69 add to $next_i$, i.e., $A = (y + D - x)u$, where $0 \leq x, y \leq D - 1$. Note that $x$ is the value of $cslot_i$ before line 68 and $y$ is the value of $cslot_i$ after line 68. Therefore, $A \in [u, (2D - 1)u]$.

By the claim's assertion, we have that $\mathit{leq}(cT_i, t_1 + u)$ holds before line 68. Since $u \leq A$, it holds that $\mathit{leq}(cT_i, t_1 + A)$ and therefore $\mathit{leq}(cT_i, t_2)$ holds.

Moreover, by the claim assertion we have that $\mathit{leq}(t_1, cT_i)$ holds. Since $A \leq (2D - 1)u$, it holds that $A - 2Du \leq -u$. This implies that $\mathit{leq}(t_1 - 2Du + A, cT_i)$. Therefore $\mathit{leq}(t_2 - 2Du, cT_i)$ holds. ■

We show that starting from an arbitrary configuration, any fair execution researches a safe configuration.

**Lemma 4.** *Let $E$ be a fair execution of the algorithm presented in Figure 2. Then, within a period of $u$, a safe configuration is reached.*

**Proof:** Let $p_i$ be a processor for which $\alpha_i$ does not hold in the starting configuration of $E$. We show that within the first complete iteration of lines 59 to 70, the predicate $\alpha_i$ holds. According to Lemma 3, all processors, $p_i$, complete at least one iteration of lines 59 to 70, within a period of $u$.

Let $a_i \in E$ be the first step in which processor $p_i$ completes the first iteration. If $\alpha_i$ does not hold in the configuration that immediately precedes $a_i$, then either (1) the predicate in line 60 holds and processor $p_i$ executes line 61 or (2) the predicate of line 64 holds at line 60.

For case (2), immediately after the execution of line 61, the predicate $\neg(\mathit{leq}(next_i - 2Du, cT_i) \wedge \mathit{leq}(cT_i, next_i))$ does not hold, because $\neg(\mathit{leq}(t - 2Du, t) \wedge \mathit{leq}(t, t))$ is false for any $t$. Moreover, the predicate in line 64 holds, since $\mathit{leq}(t, t + u)$ holds for any $t$.

In other words, the predicate in line 64 holds for both cases (1) and (2). Therefore, $p_i$ executes lines 65 to 70 in $a_i$. By Claim 5, $\alpha_i$ holds for the configuration that immediately follows $a_i$. By repeating this argument for all processors $p_i$ we show that a safe configuration is reached within a period of $u$. ■

We demonstrate the closure property of safe configurations.

**Lemma 5.** *Let $E$ be a fair execution of the algorithm presented in Figure 2 that starts in a safe configuration $c$, i.e. a configuration in which $\alpha_i$ holds for every processor $p_i$ (Definition 3). Then, every configuration in $E$ is safe with respect to LE.*

**Proof:** Let $t_i$ be the value of $p_i$'s native clock in configuration $c$ and $a_i \in E$ is the first step of processor $p_i$.

We show that $\alpha_i$ holds in configuration $c'$ that immediately follows $a_i$. Lines 73 to 76 do not change the value of $\alpha_i$. By Claim 5, if $a_i$ executes lines 65 to 70 within one complete iteration, then $\alpha_i$ holds in $c'$. Therefore, we look at step $a_i$ that includes the execution of lines 59 to 64, but does not include the execution of lines 65 to 70.

Let $t_1 = cT_i$ in $c$ and $t_2 = cT_i$ in $c'$. According to Lemma 3 and by the fairness of $E$, we have that $t_2 - t_i \bmod T < u$. Furthermore, let $A = next_i - Du$ and $B = next_i$ in $c$. The values of $next_i - Du$ and $B = next_i$ do not change in $c'$. Since $\alpha_i$ is true in $c$, it holds that $\mathit{leq}(A, t_1) \wedge \mathit{leq}(t_1, B)$. We claim that $\mathit{leq}(A, t_2) \wedge \mathit{leq}(t_2, B)$. Since $\mathit{leq}(t_1, B)$ in $c$, we have that $\mathit{leq}(t_2, B + t_2 - t_1)$ while $p_i$ executes line 64 in $a_i$. As $a_i$ does not execute lines 65 to 70 the predicate in line 64 does not hold in $a_i$. As $\mathit{leq}(t_1, B)$ and $t_2 - t_1 \bmod T < u$ the predicate in line 64 does not hold iff $\mathit{leq}(t_2, B)$. Furthermore we have that $\mathit{leq}(A, t_1)$, $\mathit{leq}(t_1, B)$, and $\mathit{leq}(t_2, B)$. As $0 < t_2 - t_1 \bmod T < u$ we have that $\mathit{leq}(A, t_2)$. Thus, $c'$ is safe as $\alpha_i$ holds in $c'$.

■

*4.3. Nice executions*

We claim that the algorithm (that is presented in Figure 2) implements nice executions. We show that every execution (for which the safe configuration requirements hold) is a nice execution.

**Theorem 1.** *Let $E$ be a legal execution of the algorithm presented in Figure 2. Then, $E$ is nice.*

**Proof:** Recall that in a legal execution all configurations are safe (Section 2). Let $a_i$ be a step in which processor $p_i$ broadcasts, $a'_i$ be the first step after $a_i$ in which processor $p_i$ broadcasts, and $a''_i$ be the first step after $a'_i$ in which processor $p_i$ broadcasts.

Let $r$, $r'$, and $r''$ be the values of $next_i$ between lines 68 and 69 in $a_i$, $a'_i$, and $a''_i$ respectively. The only changes done to $next_i$ from line 69 in $a_i$ to lines 68 and 69 in $a'_i$ are those two lines, which taken together changes $next_i$ to $next_i + Du \mod T$.

The period of length $Du$ that begins at $r$ and ends at $r' \mod T$ is divided in $D$ timeslots of length $u$. A timeslot begins at time $r + xu \mod T$ and ends at time $r + (x+1)u \mod T$ for a unique integer $x \in [0, D-1]$. The timeslot in which $a'_i$ broadcasts is *cslot* in $c$. In other words, processor $p_i$ broadcasts within a timeframe of $r$ to $r'$, which is of length $Du$. By the same arguments, we can show that processor $p_i$ broadcasts within a timeframe of $r'$ to $r''$, which is of length $Du$. These arguments can be used to show that after $a_i$, processor $p_i$ broadcasts once per period of length $Du$.

Corollary 1 considers a set of $n$ processors that broadcast once in every period of $D$ timeslots. The timeslots are of length $u$, a period that each processor estimates using its native clock. Let us consider $R$ timeframes of length $Du$. By Corollary 1, the probability that all processors successfully broadcast at least one beacon is at least $1 - 2^{-\ell}$. Now, let us consider $2R$ timeframes of length $Du$. By Corollary 1, the probability that each of the $n$ processors sends at least one beacon that all other $n-1$ processors respond to is at least $(1 - 2^{-\ell})^2 = 1 - 2^{-\ell+1} + 2^{2\ell} > 1 - 2^{-\ell+1}$. By Definition 1, $E$ is nice. ■

## 5. Performances of the algorithm

Several elements determine the precision of the clock synchronization. The clock sampling technique is one of them. Elson et al. [6] show that

the reference broadcast technique can be more precise than the round-trip synchronization technique. We allow the use of both techniques. Another important precision factor is the degree of the polynomial, $P_{i,j}(t)$, that approximates the native clock values of the neighboring processors $p_i$ and $p_j$ (see Römer et al. [17]). We consider any finite degree of the polynomial. Moreover, the clock synchronization precision improves as neighboring processors are able to sample their clocks more frequently. However, due to the limited energy reserves in sensor networks, careful considerations are required.

Let us consider the continuous operation mode. The clock precision improves as the frequency of the beacons (and responses) that the correct processors are able send increases. Thus, the precision of $P_{i,j}(t)$ depends on $round(n)$, where $round(n)$ is the time it takes $n$ processors to send $n$ beacons and then to let $n$ processors respond to all $n$ beacons.

Let us consider ideal system settings in which broadcasts never collide. Sending $n$ beacons and then letting all $n$ processors respond to each of these beacons requires the communication of at least $O(n^2)$ timestamps. By Corollary 1 and Theorem 1, we get that $2R$ timeframes of length $Du$ are needed. We also get that $R \in O(\log n)$ and $D \in O(n)$. The timeslot size $u$ is needed to fit a message with $BLog = 2R$ responses to $n$ processors. Hence, $u \in O(n \log n)$. Therefore $round(n) \in O(n^2(\log n)^2)$. Moreover, with probability that is at least $1 - 2^{-\ell+1}$, the algorithm can secure a clock precision that is $O((\log n)^2)$ times the optimum. We note that the required storage is in $O(n^2 \log n \log T)$.

## 6. Discussion

Sensor networks are particularly vulnerable to interference, whether as a result of hardware malfunction, environmental anomalies, or malicious intervention. When dealing with message collisions, message delays and noise, it is hard to separate malicious from non-malicious causes. For instance, it is hard to distinguish between a pulse delay attack from a combination of failures, e.g., a node that suffers from a hidden terminal failure, but receives an echo of a beacon. Recent studies consider more and more implementations that take security, failures and interference into account when protecting sensor networks (e.g., [11, 5]). We note that many of the existing implementations assume the existence of a fined grained synchronized clock, which we implement.

Ganeriwal et al. [9, 10] overcome the challenge of delayed beacons using the round-trip synchronization technique, and the Byzantine agreement protocol [13]. Thus, Ganeriwal et al. require $3t + 1 \leq n$. Song et al. [20] consider a different approach that uses the reference broadcasting synchronization technique. Existing statistics models refer to malicious time offsets as outliers. The statistical outlier approach is numerically stable for $2t + \epsilon \leq n \leq 3t + 1$, where $\epsilon$ is a safety constant (see [20]). We note that both approaches are applicable to our work. However, based on our practical assumptions, we are able to avoid the Byzantine agreement overheads and follow the approach of Song et al. [20]. They assume the existence of a distributed algorithm for sending beacons and collecting their responses. This work presents the first secure and self-stabilizing design of that algorithm.

The generalized extreme studentized deviate (GESD) algorithm [18] can be used to detect outliers. We note that there exists a self-stabilizing version of Song et al.'s [20] strategy. Let $B$ be the set of delivered beacon records within a period of $\mathcal{R}$. The node removes older records from $B$. The GESD algorithm tests set $B$ for outliers.

Existing implementations of secure clock synchronization protocols [23, 22, 9, 8, 14, 10, 20] are not self-stabilizing. Thus, their specifications are not compatible with security requirements for autonomous systems. In autonomous systems, the self-stabilization design criteria are imperative for secure clock synchronization. For example, many existing implementations require initial clock synchronization prior to the first pulse-delay attack (during the protocol set up). This assumption implies that the system uses global restart for self-defense management, say, using an external intervention. We note that the adversary is capable of intercepting messages continually. Thus, the adversary can risk detection and intercept all pulses for a long period. Assume that the system detects the adversary's location and stops it. Nevertheless, the system cannot synchronize its clocks without a global restart.

Sun et al. [21] describe a cluster-wise synchronization algorithm that is based on synchronous broadcasting rounds. The authors assume that a Byzantine agreement algorithm [13] synchronizes the clocks before the system executes the algorithm. Our algorithm is comparable with the requirements of autonomous systems and makes no assumptions on synchronous broadcasting rounds or start.

Manzo et al. [14] describe several possible attacks on an (unsecured) clock synchronization algorithm and suggest counter measures. For single hop synchronization, the authors suggest using a randomly selected "core" of

nodes to minimize the effect of captured nodes. The authors do not consider the cases in which the adversary captures nodes after the core selection. In this work, we make no assumption regarding the distribution of the captured nodes. Farrugia and Simon [8] consider a cross-network spanning tree in which the clock values propagate for global clock synchronization. However, no pulse-delay attacks are considered. Sun et al. [22] investigate how to use multiple clocks from external source nodes (e.g., base stations) to increase the resilience against an attack that compromises source nodes. In this work, there are no source nodes.

In [23], the authors explain how to implement a secure clock synchronization protocol. Although the protocol is not self-stabilizing, we believe that some of their security primitives could be used in a self-stabilizing manner when implementing our self-stabilizing algorithm.

Herman and Zhang [12] present a self-stabilizing clock synchronization algorithm for sensor networks. The authors present a model for proving the correctness of synchronization algorithms and show that the converge-to-max approach is stabilizing. However, the converge-to-max approach is prone to attacks with a single captured node that introduces the maximal clock value whenever the adversary decides to attack. Thus, the adversary can at once set the clock values "far into the future", preventing the nodes from implementing a continuous time approximation function. This work is the first in the context of self-stabilization to provide security solutions for clock synchronization in sensor networks.

### *6.1. Conclusions*

Designing secure and self-stabilizing infrastructure for sensor networks narrows the gap between traditional networks and sensor networks by simplifying the design of future systems. In this work, we consider realistic system settings and take a clean slate approach in designing a fundamental component; a clock synchronization protocol.

The designers of sensor networks often implement clock synchronization protocols that assume the system settings of traditional networks. However, sensor networks often require fine-grained clock synchronization for which the traditional protocols are inappropriate.

Alternatively, when the designers do not assume traditional system settings, they turn to reinforce the protocols with masking techniques. Thus, the designers assume that the adversary never violates the assumptions of the masking techniques, e.g., there are at most $t$ captured nodes at all times,

where $3t+1 \leq n$. Since sensor networks reside in an unattended environment, the last assumption is unrealistic.

Our design promotes self-defense capabilities once the system returns to following the original designer's assumptions. Interestingly, the self-stabilization design criteria provide an elegant way for designing secure autonomous systems.

*6.2. Acknowledgments*

This work would not have been possible without the contribution of Marina Papatriantafilou in many helpful discussions, ideas, and analysis. We wish to thank Ted Herman for many helpful discussions. Many thanks to Edna Oxman for improving the presentation.

## References

[1] N. Abramson et al. *The Aloha System*. Univ. of Hawaii, 1972.

[2] Murat Demirbas, Anish Arora, Tina Nolte, and Nancy A. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In Teruo Higashino, editor, *OPODIS*, volume 3544 of *LNCS*, pages 299–315. Springer, 2004.

[3] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[4] Shlomi Dolev. *Self-Stabilization*. MIT Press, March 2000.

[5] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport. Gossiping in a multi-channel radio network. In Andrzej Pelc, editor, *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[6] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *Operating Systems Review (ACM SIGOPS)*, 36(SI):147–163, 2002.

[7] Jeremy Elson, Richard M. Karp, Christos H. Papadimitriou, and Scott Shenker. Global synchronization in sensornets. In Martin Farach-Colton, editor, *LATIN*, volume 2976 of *LNCS*, pages 609–624. Springer, 2004.

[8] Emerson Farrugia and Robert Simon. An efficient and secure protocol for sensor network time synchronization. *J. Syst. Softw.*, 79(2):147–162, 2006.

[9] Saurabh Ganeriwal, Srdjan Capkun, Chih-Chieh Han, and Mani B. Srivastava. Secure time synchronization service for sensor networks. In *Proceedings of the 4th ACM workshop on Wireless security (WiSe'05)*, pages 97–106, NYC, NY, USA, 2005. ACM Press.

[10] Saurabh Ganeriwal, Srdjan Capkun, and Mani B. Srivastava. Secure time synchronization in sensor networks. *ACM Transactions on Information and Systems Security*, March 2006.

[11] Seth Gilbert, Rachid Guerraoui, and Calvin C. Newport. Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In Alexander A. Shvartsman, editor, *OPODIS*, volume 4305 of *LNCS*, pages 215–229. Springer, 2006.

[12] Ted Herman and Chen Zhang. Best paper: Stabilizing clock synchronization for wireless sensor networks. In Ajoy Kumar Datta and Maria Gradinariu, editors, *SSS*, volume 4280 of *LNCS*, pages 335–349. Springer, 2006.

[13] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[14] Michael Manzo, Tanya Roosta, and Shankar Sastry. Time synchronization attacks in sensor networks. In *Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks (SASN'05)*, pages 107–116, NYC, NY, USA, 2005. ACM Press.

[15] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[16] Kay Romer. Time synchronization in ad hoc networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 173–182, NYC, NY, USA, 2001. ACM Press.

[17] Kay Römer, Philipp Blum, and Lennart Meier. Time synchronization and calibration in wireless sensor networks. In Ivan Stojmenovic, editor, *Handbook of Sensor Networks: Algorithms and Architectures*, pages 199–237. John Wiley and Sons, Sep. 2005.

[18] B. Rosner. Percentage points for a generalized *esd* many-outlier procedure. *Technometrics*, 25:165–172, 1983.

[19] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 2nd edition, 1996.

[20] Hui Song, Sencun Zhu, and Guohong Cao. Attack-resilient time synchronization for wireless sensor networks. *Ad Hoc Networks*, 5(1):112–125, 2007.

[21] Kun Sun, Peng Ning, and Cliff Wang. Fault-tolerant cluster-wise clock synchronization for wireless sensor networks. *IEEE Transactions on Dependable and Secure Computing*, 2(3):177–189, 2005.

[22] Kun Sun, Peng Ning, and Cliff Wang. Secure and resilient clock synchronization in wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 24(2):395–408, Feb. 2006.

[23] Kun Sun, Peng Ning, and Cliff Wang. Tinysersync: secure and resilient time synchronization in wireless sensor networks. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 264–277. ACM, 2006.