



Secure and self-stabilizing clock synchronization in sensor networks

Jaap-Henk Hoepman^{a,b}, Andreas Larsson^c, Elad M. Schiller^{c,*}, Philippas Tsigas^c

^a TNO ICT, P.O. Box 1416, 9701 BK Groningen, The Netherlands

^b Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

^c Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, Rännvägen 6B SE-412 96 Göteborg, Sweden

ARTICLE INFO

Keywords:

Secure and resilient computer systems
Sensor-network systems
Clock-synchronization
Self-Stabilization

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 an efficient clock sampling algorithm which tolerates attacks by this adversary, collisions, a bounded amount of losses due to ambient noise, and a bounded number of captured nodes that can jam, intercept, and send fake messages. The algorithm is self-stabilizing, so if these bounds are temporarily violated, the system can efficiently stabilize back to a correct state. Using this clock sampling algorithm, we construct the first self-stabilizing algorithm for secure clock synchronization in sensor networks that is resilient to the aforementioned adversarial attacks.

© 2010 Elsevier B.V. All rights reserved.

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 [3]). 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 capture nodes and 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 ambient noise.

The core of our clock synchronization algorithm is a mechanism for sampling the clocks of neighboring nodes in the network. Of especial importance is the sampling of clocks at reception of broadcasts called beacons. A beacon acts as a shared reference point because nodes receive it at approximately the same time (propagation delay is negligible for these radio transmissions). Elson et al. [9] use such samples to approximate the clocks of neighboring nodes. They use linear regression

* Corresponding author. Tel.: +46 736439754.

E-mail addresses: jaap-henk.hoepman@tno.nl (J.-H. Hoepman), larandr@chalmers.se (A. Larsson), elad@chalmers.se (E.M. Schiller), tsigas@chalmers.se (P. Tsigas).

to deal with differences in clock rates. The basic algorithm synchronizes a cluster. Overlapping clusters with shared gateway nodes can be used to convert timestamps among clusters. Karp et al. [15,16] input clock samples of beacon receipts into an iterative algorithm, based on resistance networks, to converge to an estimated global time. Römer et al. [24] give an overview of methods that use samples from other nodes to approximate their clocks. They present phase-locked looping (PLL) as an alternative to linear regression and present methods for estimating lower and upper bounds of neighbors' clocks. Note that none of these articles takes security or self-stabilization into account.

As mentioned above, 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 [11,12,28] and Section 2.3.1). We are interested in fine-grained clock synchronization, where there are no cryptographic countermeasures 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 [26]). 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 could 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., [26,31]). However, we assume that the adversary can *capture* nodes, reveal their entire state (including private variables), stop their execution, and impersonate them. The adversary can also lead them to send erroneous information and launch jamming (or collision) attacks.

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. [13], which consider the minimal requirements for message delivery under broadcast interception attacks.) We explain how to sift out responses to delayed beacons by following the above assumptions that consider many practical issues.

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. 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 [4,5] cope with the occurrence of transient faults in an elegant way. Bad configurations might occur due to the occurrence of an arbitrary combination of failures. Self-stabilizing systems can be started in *any* configuration. 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. Uncaptured nodes behave correctly at all times. Furthermore, the communication model is fair. It resembles that of [2] and does not consider Byzantine behavior in the communication medium. However, captured nodes can behave in a Byzantine manner at the processor level. We design a distributed algorithm for sampling the clocks of g neighboring nodes in the presence of f captured and/or pulse-delay attacked nodes. Although captured nodes remain captured, a node whose pulse-delay attacked messages are no longer in the buffer of any uncaptured node will not count toward f anymore. We focus on captured nodes and delay attacks, but f can be extended to include nodes with timing failures and other ways of not following protocol.

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. For example, [12] uses Byzantine agreement (this requires $3f + 1 \leq g$), and [28] considers the statistical outliers (this requires $2f + O(1) \leq g$). (See Section 7 for details on the masking techniques.) Although Byzantine agreement is one possible filtering technique, we do not consider Byzantine faults, as stated above.

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 [23]).

1.1. Our contribution

We present the first design for secure and self-stabilizing clock synchronization in sensor networks resilient to an adversary that can capture nodes and launch pulse-delay attacks. The core is a secure and self-stabilizing algorithm for sampling clocks of neighboring nodes.

The algorithm secures, with high probability, sets of complete neighborhood clock samples with a period that is $O((\log n)^2)$ times the optimum. The optimum requires, in the worst case, the communication of at least $O(n^2)$ timestamps. Here n is a bound on the number of sensor nodes that can interfere with a node (potentially the number of nodes within

transmission range of the node). It is of high importance for high-precision clock synchronization that the clock sampling period is small since the offsets and frequencies of the nodes' clocks change over time.

Our design tolerates transient failures that may occur due to temporary violation of the designer's assumption. For example, the number of captured and/or pulse-delay attacked nodes could exceed more than f and then sink below f (delayed messages eventually vanish from queues). After the system resumes operation according to the designer's assumption, the system will stabilize within one communication timeslot (that is of size $O(n \log n)$). We assume that (before and after the system's recovery) there are message omission failures, say, due to ambient noise, attacks or the algorithm's message collisions.

The correct node sends beacons and responds to the other nodes' beacons. We use a randomized strategy for beacon scheduling that guarantees regular message delivery with high probability.

1.2. Document structure

We start by describing the system settings (Section 2) and formally present the algorithm (Section 3). A description of our execution system model (Section 4) and a proof of the algorithm correctness (Section 5) are followed by a performance evaluation (Section 6). Then we review the literature and draw our conclusions (Section 7).

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 . In addition, we assume that every processor $p_i \in P$ has a unique identifier, i . A processor identifier can be represented by a known and fixed number of bits in memory. In that respect there is a known upper bound on the number of processors.

2.1. Time, clocks, and their notation

We follow settings that are compatible with those of Herman and Zhang [14]. 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; *logical time* builds on native time with an additive adjustment factor. This factor is adjusted to approximate a shared clock, whether local to a neighborhood or global to the entire network.¹

We consider applications that require the clock interface to include the *read* operation, which returns a *timestamp* with T possible states.² Let $C^i(t)$ denote the value $p_i \in P$ gets from a *read* of the native clock at real time t .

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 logical clock in order to achieve synchronization, but never adjusts the native clock. We define the native clock *offset* between any two processors p_i and p_j as $\delta_{i,j}(t) = C^i(t) - C^j(t)$. We assume that, at any given time, the native clock offset is arbitrary. Moreover, the *skew* of p_i 's native clock, ρ_i , is the first derivative of the clock value with respect to real time. Thus $\rho_i = \lim_{\tau \rightarrow 0} (C^i(t + \tau) - C^i(t)) / \tau$. We assume that $\rho_i \in [\rho_{\min}, \rho_{\max}]$ for any processor p_i , where $\rho_{\min} = 1 - \kappa$ and $\rho_{\max} = 1 + \kappa$ are known constants, 1 is the real time unit and $\kappa \geq 0$. The second derivative of the clock's offset is called *drift*. We allow non-zero drift as long as $\rho_i \in [\rho_{\min}, \rho_{\max}]$.

2.2. Communications

Wireless transmissions are subject to collisions 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.

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 by $G_i \subseteq \{p_j \in P \mid R_{lb} \geq |L_i - L_j|\}$. Furthermore, we denote the set of processors that can interfere with the communications of p_i by $\vec{G}_i \subseteq \{p_j \in P \mid 2R_{lb} \geq |L_i - L_j|\}$. We note that G_i is not something processor p_i needs to know in advance, but something it discovers as it receives messages from other processors.

A successful broadcast by a processor p_i occurs when the message is received by all other processors in G_i . A successful broadcast to a set $K \subseteq G_i$ occurs when the message is received by all other processors in K .

We assume that $n \geq |\vec{G}_i|$ for any processor p_i . In other words, n is a known upper bound on the number of nodes that can interfere with any one node's communication (including that node itself). In the worst-case scenario $G_i = \vec{G}_i$ and thus

¹ Lenzen et al. [19,18] and Sommer and Wattenhofer [27] also refer to the term of logical time as "logical clock values". Herman and Zhang [14] refer to it as local time and build global time on top of the local time. See Section 7.

² In footnote 6 we show what the minimal size of T is.

potentially $|G_i| = n$. Furthermore, a node will receive information from neighbors about their neighbors, so in the worst-case scenario a node needs to keep track of data about n nodes. For simplicity we therefore use n as a bound of the number of neighbors (including the node itself) as well. This does not mean that we only consider a cluster of n nodes.

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 queue 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. It includes both transmission delay and propagation delay, even though the propagation delay is negligible in comparison with the transmission delay.

We associate each $LBcast$ and $LBrecv$ operation with a native clock timestamp for the moment of sending and receiving. We assume the existence of an efficient algorithm for timestamping a message in transfer and a message being received as close to the physical layer as possible (see [31]).

2.2.2. The environment

Messages might be lost to ambient noise as well as collisions of the nodes' transmissions. Collisions due to attacks made by the adversary or by captured nodes are called *adversarial collisions*. Message collisions due to concurrent transmissions of nodes that follow the message scheduling of the algorithm are called *non-adversarial collisions*. A broadcast that is not lost due to ambient noise or adversarial collisions is said to be *fair*. We note that a fair broadcast can still be lost due to non-adversarial collisions.

The environment can execute the operation $omission(m_i)$ (which is associated with a particular message, m_i , sent by processor p_i) immediately after $LBcast_i(m_i)$. The environment selects a (possibly empty) subset of p_i 's neighbors ($K_i \subseteq G_i$) and removes any message m_i from their queues $queue_{i,j}$ (such that $p_j \in K_i$).

Below we talk about what “the environment” selects when it comes to message omission. Here we see the environment as a global adversary, separate and independent from the “regular” malicious and locally bound adversary of Section 2.3. The term “adversary” is only used for that “regular” malicious adversary.

When a processor p_i and a processor $p_j \in \vec{G}_i$ do concurrent broadcasts of messages m_i and m_j we assume that the environment arbitrarily selects $K_i \subseteq G_i \cap G_j$ when invoking $omission(m_i)$ due to the collision (and vice versa for m_j). For details on what it means in our execution system model see Section 4.3. In other words, when two processors with overlapping communication ranges broadcast concurrently, there are no guarantees of delivery, for those messages, within the overlap (regardless of noise). This is a simple and general model for message collisions. It is possible to let a more specialized physical layer model resolve the subset K_i .

The environment selects messages to omit due to ambient noise as described at the end of Section 2.2.3. The adversary selects messages to omit due to omission attacks as described at the end of Section 2.3.1.

2.2.3. Ambient noise

The parameter $\xi \geq 1$ denotes the maximal number of repeated transmissions required (by any particular processor) to get at least one fair broadcast. Such a broadcast can still be lost due to non-adversarial collisions. These assumptions model the ambient noise of the communication channel, as well as omission attacks by the adversary and by captured nodes (see Section 2.3.2). Furthermore, we assume that all processors know ξ .

The environment selects messages to remove due to ambient noise, but is limited by ξ as described above. We assume that the choice of messages omitted due to ambient noise is independent from the choice of messages omitted due to non-adversarial collisions.

2.3. The adversary

We assume that there is a single adversary. The goal of the adversary is to disturb the clock synchronization algorithm so that clock samplings become erroneous, or even misleading. At the same time, the adversary does not want to let its presence be known by launching obvious attacks.

2.3.1. Omission attacks and delay attacks

The adversary can launch omission and delay attacks against a message sent by another processor. We assume that at any time the adversary, just like all processors, has a distinct (unknown) location in space. We assume that the adversary's 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 a beacon's broadcast is omitted or delayed.

Consider a message, m_i , broadcast by a processor, p_i , and attacked by the adversary. We assume that the adversary chooses a sphere with its own location in the center. We denote the set of processors within the sphere S . The nodes in $S \cap G_i$ will be affected by the attack against m_i .

The adversary launches message omission attacks (also known as interception attacks) by jamming the medium. The environment invokes $omission(m_i)$ for all processors in $S \cap G_i$. This selection is limited by the assumptions regarding ξ , as described in Section 2.3.2.

For delay attacks, we follow the model of Ganeriwal et al. [11,12]. The adversary can receive (at least part of) a message, jam the medium for a set of nodes before they receive it in whole, and then replay the message slightly later. The adversary resends the message to the processors in $S \cap G_i$ after a chosen delay. The resent message is potentially lost due to ambient noise or collisions, like any other message. The processors in $S \cap G_i$ that receive m_i thus receive it later than they normally would have.

Other ways to do delay attacks include considering an adversary with directional antennas (which we do not consider) sending the same message at slightly different times in different directions, or having a captured node sending a message within a smaller radius and having the adversary repeating that within an area that was left out (see [28] for details). Both these delay attacks require the delayed message to originate from the adversary impersonating a captured node or from a captured node. We make the weaker assumption that a message from any processor can, potentially, be delayed by the adversary.

2.3.2. Omission attack limitations

We let ξ (see Section 2.2.3) include ambient noise as well as collisions deliberately produced by the adversary and by captured nodes. The adversary or the captured nodes could jam the medium such that the assumption of ξ does not hold. If too many messages are lost, however, that can act as an alarm that an adversary is present. This is something that the adversary, who wants to go undetected, wants to avoid. Furthermore, if the adversary totally jams the communication medium, clock synchronization will not take place. As a result, the adversary has no possibility to directly influence the logical clock. Thus, this is not an option for an adversary that wants to manipulate tracking algorithms to present a misleading view of its whereabouts and movements.

We note that the adversary cannot predict the broadcasting schedule of uncaptured nodes. Thus, adversarial collisions, covered by ξ (together with ambient noise), are independent from non-adversarial collisions.

Gilbert et al. [13] consider the minimal requirements for message delivery under broadcast interception attacks. They assume that the adversary intercepts no more than β broadcasts of the algorithm, where β is an unknown constant that reflects the maximum amount of energy an adversary wants to use for disruption of communications. 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 α messages and the adversary can intercept at most β of the algorithm's messages. Our system settings are comparable to the assumptions made by Gilbert et al. [13] on the ratio of β/α . However, in contrast to the unknown β , we assume that the maximum ratio is a known constant that reflects the maximum amount of disruption the adversary can get away with, without being detected.

2.3.3. Captured nodes

The adversary can capture nodes by moving to their location and accessing them physically. For any processor p_i , we assume that the number of captured and/or pulse-delay attacked nodes is no more than f , within its neighborhood, G_i . Here, f depends on $|G_i|$ and the filtering mechanism that is being used. (For example, $3f + 1 \leq |G_i|$ for the Byzantine agreement masking technique as in [12] and $2f + \epsilon \leq |G_i|$ for the outlier masking technique as in [28]; see Section 7 for more details.)

When the adversary captures a processor p_i , the adversary gains all information contained in the processor's memory, like secret keys, seeds for pseudorandom generators, etc. The adversary can lead a captured processor p_i to send incorrect data to processors in G_i . It can also lead the captured node to jam the communication media with noise or with collisions among processors in G_i . The set of target processors are further limited to a sphere with the captured node in the center (cf. the sphere limitation for attacks launched directly by the adversary, in Section 2.3.1.) These noise and collision attacks are also limited by ξ as described in Section 2.3.2, just like attacks launched directly by the adversary.

2.3.4. 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., [26,31]). 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.

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 objective of the clock synchronization protocol is (1) to sample the clocks of its neighbors by 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 logical clock after sifting out responses to delayed beacons. Unlike objectives (1) to (3), the clock estimation task is not a hard real-time task. Therefore, the algorithm outputs records to the upper layer that synchronizes the logical clock after neutralizing the effect of pulse-delay attacks (see Section 7 for details on techniques for filtering out delayed messages). 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 can deliver to the upper layer the records of the 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 noise and message collisions. The celebrated Aloha protocol [1] (which does not consider nondeterministic fluctuating skews) inspires us to take a randomized strategy for scheduling broadcasts. We overcome the difficulties above and show 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, ℓ , 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* [11,12] and *reference broadcasting* [9]. For example, in the round-trip synchronization technique, the sender p_j sends a timestamped message $\langle t_1 \rangle$ to receivers, $p_k \in G_j$, 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 $\langle j, t_1, \{ \langle k, \langle t_2, t_3, t_4 \rangle \} \rangle$, where $\{ \langle k, \langle t_2, t_3, t_4 \rangle \} \}$ 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, $\langle t_r^j \rangle$, that p_i sends to node $p_j \in G_i$ for p_j 's previous beacon, t_s^j , where t_r^j is the time in which p_i received p_j '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 $\langle t_s^i, \langle t_s^{j_1}, t_r^{j_1} \rangle, \langle t_s^{j_2}, t_r^{j_2} \rangle, \dots \rangle$, which includes all processors $p_{j_k} \in G_i$.

Now, suppose that the schedules are not done in a Round Robin fashion. We denote p_i 's sequence of up to $BLog$ most recently sent beacons with $\langle t_s^i(k) \rangle_{0 \leq k < BLog}$, among which $t_s^i(k)$ 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 $p_j \in G_i$ 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, p_i stores its last $BLog$ beacon messages in a FIFO queue, $q_i[k] = \langle t_s^i(k) \rangle_{0 \leq k < BLog}$. Moreover, every beacon message includes all responses to the $BLog$ most recently received beacons from all nodes. Let $q_j = \langle q_j[k] \rangle_{0 \leq k < BLog}$ be p_i 's FIFO queue of the last $BLog$ records of the form $\langle t_s^j(k), t_r^j(k) \rangle$, 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 $\langle q_i, q_{j_1}, q_{j_2}, \dots \rangle$, which includes all processors $p_{j_k} \in G_i$. 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 Fig. 2, includes two procedures: (1) a do-forever loop that schedules and broadcasts beacon messages (lines 66 to 80) and (2) an upon message arrival procedure (lines 82 to 87).

3.2.1. The do-forever loop

The do-forever loop periodically tests whether the "timer" has expired (in lines 67 to 74).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 69). The algorithm then removes data, gathered by p_i itself, that are too old (lines 70 to 71). (Note that under normal circumstances,

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

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.)

the data never become too old before they are pushed out by new data at line 77 or line 86). The algorithm then tests that all the stored data (including data received from others) are ordered and timely (line 72). Timely here means that timestamps collected by a processor p_j is not too old or in the future compared to the latest time of p_j 's native clock, that p_i has received. In the case where the recorded information about beacon messages is incorrect, the algorithm flushes the queues (line 73). The data received by others are tested at line 72 in the same way as at reception (line 83). Data that do not pass the test at line 83 are never stored. Therefore, if the buffers are flushed it is due to internal data corruption (in the starting configuration), and not due to receipt of bad data (during execution). We note that transient faults can be the source of such internal data corruption. However, bad data may be received (and therefore rejected) at any time during the execution, say, from captured nodes.

When the timeslot arrives, the processor outputs a synchronizer case record for the oldest beacon, in the queue with its own beacons (line 76). It contains for each of the other processors, $p_j \in G_i$, the receive time of that beacon. Moreover, it contains, for processor p_j , the send and receive times for a later 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 77). The next schedule for processor p_i is set (lines 78 and 79) just before it broadcasts the beacon message (line 80).

3.2.2. The message arrival

When a beacon message arrives (line 82), processor p_i gets j , the id of the sender of the beacon, r , p_i 's native time at the receipt of the beacon, and v , the message of the beacon. The algorithm sanity checks the received data (line 83). If they are ordered and timely (not too old or in the future compared to the latest timestamp from p_j) the data are processed (lines 84 to 87). Otherwise the message is ignored.

Passing the sanity check, processor p_i then outputs a record of the non-synchronizer case (lines 84 to 85). These data can be used for the reference broadcast technique in the upper layer. It finds the oldest beacon in the queue with data on beacons received by p_j . The record contains responses from processors $p_k \in G_j$ that refer to this beacon. Furthermore, it contains data about later messages back, from the receiving processors p_k to processor 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 86) and the message itself (line 87).

4. Execution system model

4.1. 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 [5]), 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.⁵

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, \dots) , where each s_j is the state of processor p_j (including messages in transit for p_i). We define an *execution* $E = c[0], a[0], c[1], a[1], \dots$ 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 (see Fig. 1).

4.2. Tracing timestamps and communications

As stated in Section 2.2.1, we associate each *LBcast* and *LBrecv* operation with a timestamp for the moment of sending and receiving. The timestamp of an *LBcast* operation is the native time at which message m is sent, and this information is included in the sent message. When processor p_i executes the *LBrecv* operation, an event is triggered with the arguments j , t , and $\langle m \rangle$: $p_j \in G_i$ is the sending processor of message $\langle m \rangle$, which p_i receives when p_i 's native clock is t . We note that

⁵ We note that a simple random scheduler can be used for the case in which time t does not follow a uniform distribution.

Constants:		$\ell =$ tuning parameter (see Corollary 1)
2	$i =$ id of executing processor	8 $BLog = 2 \lceil \xi \frac{\ell + \log_2(\lceil \frac{\rho_{max}}{\rho_{min}} \rceil + 1)n}{-\log_2(1-1/e)} \rceil$, backlog size
	$n =$ bound on # of interfering processors (incl. itself)	$D = 3n (\lceil \frac{\rho_{max}}{\rho_{min}} \rceil + 1)$, the broadcast timeslots
4	$w =$ compensation time between lines 67 and 80	10 $T =$ number of possible states of a timestamp
	$d =$ upper bound on message communication delay	$\rho_{min} =$ lower bound on clock skew
6	$u =$ size of a timeslot in time units ($u > d + w$)	12 $\rho_{max} =$ upper bound on clock skew
<hr/>		
Variables:		
14	$m[n] =$ all received messages and timestamp	20 $native_clock :$ immutable storage of the native clock
16	each entry is an array $v[n]$	$cslot : [0, D-1] =$ current timeslot in use
	each entry is a queue $q[BLog]$	22 $next : [0, T-1] =$ schedule of next broadcast
18	each entry is a pair $\langle s, r \rangle$	$cT =$ last do-forever loop's timestamp
<hr/>		
External functions:		
26	$output(R) :$ delivers record R to the upper layer	32 $first(Q) :$ least recently enqueued element in Q , number 0
	$choose(S) :$ uniform selection of an item from the set S	$last(Q) :$ most recently enqueued element in Q
28	$keys(v) :$ the set of id:s that indexes v	34 $full(Q) :$ whether queue Q is full
	$enqueue(Q) :$ adds an element to the end of the queue Q	$flush(Q) :$ empties the queue Q
30	$dequeue(Q) :$ removes the front element of the queue Q	36 $get_s(Q) :$ list elements of field s in Q
	$size(Q) :$ size of the queue Q	$get_r(Q) :$ list elements of field r in Q
<hr/>		
Macros and inlines:		
40	$border(t) : (D-cslot)u + t \bmod T$	
	$schedule(t) : cslot-u + t \bmod T$	
42	$leq(x, y) : (\exists b : 0 \leq b \leq 2 BLog D u \wedge y \bmod T = x + b \bmod T)$	
	$enq(q, m) : \{ \text{while } full(q) \text{ do } dequeue(q); enqueue(m) \}$	
44	$G(j) : keys(m[j].v)$	
46	$expire_s(q, t) : (* Expires data based on send times *)$	
	while $size(q) > 0 \wedge leq(first(q), s, t)$ do	
48	dequeue(q)	
	$expire_r(q, t) : (* Expires data based on receive times — as expire_s but with .r instead of .s *)$	
50	$check() : \wedge \{ checkdata(m[j].v, j) : j \in keys(m[i].v) \}$	
	$checkdata(v, j) : (* Coherency test for data from processor j *)$	
52	$\wedge \{ checklist(get_s(v[k].q), lclock(v, j)) \wedge (j = k \vee checklist(get_r(v[k].q), lclock(v, j))) : k \in keys(v) \}$	
	$checklist(q, t) : (* Checks that all elements of a list are chronologically ordered and not in the future *)$	
54	$size(q) = 0 \vee (leq(first(q), t) \wedge leq(last(q), t) \wedge \{ leq(q[b_1].q, q[b_2]) : b_1 < b_2, \{b_1, b_2\} \subseteq [1, size(q)] \})$	
	$lclock(v, j) : last(v[j].q).s$	
56	$(* Get response-record for p_k, for p_j as the synchronizer *)$	
58	$ts(s, j, k) : \{ \text{if } (\exists b_1^j, b_2^j, b_1^k, b_2^k :$	
	$s = m[j].v[j].q[b_1^j].s = m[k].v[j].q[b_1^k].s \wedge$	
60	$m[k].v[k].q[b_2^k].s = m[j].v[k].q[b_2^j].r \wedge$	
	$leq(m[j].v[j].q[b_1^j].s, m[j].v[k].q[b_2^j].r) \wedge$	
62	$leq(m[k].v[j].q[b_1^k].r, m[k].v[k].q[b_2^k].s) \}$	
	then return $\langle m[k].v[j].q[b_1^k].r, m[k].v[k].q[b_2^k].s, m[j].v[k].q[b_2^j].r \rangle$	
64	else return \perp }	

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

66	Do forever, every $u/2$	82	Upon $LBrecv(j, r, v)$ (* $i \neq j$ *)
	let $cT = read(native_clock) + w$		if $checkdata(v, j)$ then
68	if $\neg (leq(next-2Du, cT) \wedge leq(cT, next+u))$ then	84	let $s = first(m[i].v[j].q).s$
	$next \leftarrow cT$		output $\langle j, s, \{ \langle k, ts(s, j, k) \rangle : k \in G(j) \setminus \{j\} \} \rangle$
70	$expire_s(m[i].v[i].q, cT)$	86	$enq(m[i].v[j].q, \langle last(v[j].q).s, r \rangle)$
	$\forall j \in G(i) \setminus \{i\}$ do $expire_r(m[i].v[j].q, cT)$		$m[j].v \leftarrow v$
72	if $\neg check()$ then		
	$\forall j, k \in P$ do $flush(m[j].v[k].q)$		
74	if $leq(next, cT) \wedge leq(cT, next + u)$ then		
	let $s = first(m[i].v[i].q).s$		
76	output $\langle i, s, \{ \langle j, ts(s, i, j) \rangle : j \in G(i) \setminus \{i\} \} \rangle$		
	$enq(m[i].v[i].q, \langle cT, \perp \rangle)$		
78	$(next, cslot) \leftarrow (border(next), choose([0, D-1]))$		
	$next \leftarrow schedule(next)$		
80	$LBcast(m[i])$		

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

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.

4.3. Concurrent versus 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 $p_j \in \vec{G}_i$ 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_k^r 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, as described in Section 2.2.2.

4.4. Fair executions

We say that execution E has *fair communications*, if, whenever processor p_i broadcasts ξ successive messages (successive in terms of the algorithm’s messages sent by p_i), at least one of these broadcasts is fair, i.e., not lost to noise or adversarial collisions. We note that fair communication does not imply reliable communication even for $\xi = 1$, because a message can still be lost due to non-adversarial collisions. 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).

4.5. 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.

5. Correctness

In this section we demonstrate that the task of random broadcast scheduling is achieved by the algorithm that is presented in Fig. 2. Namely, with high probability, the scheduler allows the exchange of beacons and responses within a short time. The objectives of the random broadcast scheduling task are defined in Definition 1 and consider *broadcasting rounds*. To consider a number of broadcasting rounds from a point in time (such as the time associated with a step a), is to consider the time needed for every processor to fit in that many partitions, i.e., broadcast that many times.

Definition 1 (Nice Executions). Let us consider the executions of the algorithm presented in Fig. 2. Furthermore, let us consider a processor p_i and let Γ_i be the set of all execution prefixes, E_{Γ_i} , such that, within the first \mathcal{R} broadcasting rounds of E_{Γ_i} , (1) every processor $p_j \in G_i$ (including p_i) successfully broadcasts at least one beacon to all processors $p_k \in G_i \cap G_j$ and (2) every such processor p_j gets at least one response from all such processors p_k . We say that execution E is *nice* in relation to processor p_i if E has a prefix in Γ_i .

The proof of Theorem 1 (Section 5.3, page 30) demonstrates that, when considering $\mathcal{R} = 2R$, for any processor p_i , the algorithm reaches a nice execution in relation to p_i with probability of at least $1 - 2^{-\ell+1}$, where ℓ 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 $p_j \in G_i$ (when considering the neighborhood of any processor p_i) to each broadcast at least one message that is received by all other processors in $G_i \cap G_j$.⁶

Once the system reaches a nice execution in relation to a processor p_i , and the exchange of beacons and responses occurs, the following holds. 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 beacon record, rec_k , which 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 beacons from all processors $p_j \in G_i$. Furthermore, for each such beacon from a processor $p_j \in G_i$, p_i gathers responses to those beacons from all processors $p_k \in G_i \cap G_j$. For this reason, we set $BLog$ to be \mathcal{R} .

5.1. 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

⁶ 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.

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.

Messages from processor p_i can collide with messages from up to $n - 1$ other processors if $|\vec{G}_i| = n$. Furthermore, in the worst-case scenario $|G_i| = |\vec{G}_i| = n$ for processor p_i . We want to guarantee with high probability that within G_i everyone exchanges messages. Therefore, we look at n players throwing balls into bins when analyzing the message scheduling algorithm. Our results will also hold for cases when $|G_i| < n$ and when $|\vec{G}_i| < n$, as the probability of collisions in those cases is equal to or lower than that for the worst-case scenario.

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 (or omission attacks). 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 (Lemmas 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 (and omission attacks) (Claim 4) and conclude the analysis by considering general system settings (Corollary 1).

5.1.1. 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 broadcast 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.

5.1.2. 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 $x \geq 2$ it holds that

$$\left(1 - \frac{1}{x}\right)^{x-1} > \frac{1}{e}. \quad (1)$$

Proof. It is well known that

$$\left(1 + \frac{1}{x}\right)^x < e \quad (2)$$

for any $x \geq 1$. From this it follows that

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

for $x \geq 2$. \square

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

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

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 being 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 being 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 get 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 [21]:

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

The random variables assume only non-negative values. Markov's Inequality [21], $\Pr(X \geq a) \leq E[X]/a$, therefore gives us

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

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

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

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.

5.1.3. 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 hits 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 5.1.1). 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_k^i and $b_{k'}^j$, respectively, such that both b_k^i and $b_{k'}^j$ overlap. The bin b_k^i can either overlap with the bins $b_{k'-1}^j$ and $b_{k'}^j$ or (exclusively) overlap with the bins $b_{k'}^j$ and $b_{k'+1}^j$. Balls aimed at any of the bins possibly overlapping with b_k^i (namely $b_{k'-1}^j$, $b_{k'}^j$ and $b_{k'+1}^j$) are regarded as colliding with the ball of player \hat{p}_j . The same argument applies to bin $b_{k'}^j$ overlapping with bins b_{k-1}^i , b_k^i and b_{k+1}^i . 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 the 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, this 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 being successful does not depend on how other players partition the bins. \square

5.1.4. 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 being unsuccessful in a partition. By taking $R_{\text{skew}} = R \geq (\ell + \log_2 \hat{\rho}n)/(-\log_2 p) \in O(\ell + \log(n))$, 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**,

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

the guarantees of **Lemma 2** hold. \square

5.1.5. Communication channels with noise

In our system settings, message loss occurs due to noise and omission attacks and not only due to the algorithm’s message collisions. Recall that ξ defines the number of broadcasts required in order to guarantee at least one fair broadcast (not lost to noise or adversarial collisions; see Section 2.2.3). 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 ξ consecutive throws. Omission attacks are incorporated into the ξ assumption and are thus not seen as a ball being thrown.

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 ξR partitions instead of R partitions. This will guarantee that each player gets at least R “fair” balls. That is, each player gets at least R balls that are either successful or that bounce out due to collision with another ball. Thus, the asymptotic number of bins is unchanged and the guarantees of Lemma 2 still hold. \square

5.1.6. 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 2–4.

Corollary 1. Suppose that every processor broadcasts once in every partition of D timeslots. Consider any processor p_i . The probability that every processor $p_j \in G_i$ successfully broadcasts at least one beacon to every processor $p_k \in G_i \cap G_j$ within R partitions is at least $1 - 2^{-\ell}$ when

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

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

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

$$p = 1 - \frac{1}{e}. \tag{16}$$

Corollary 1 shows that, within a logarithmic number of broadcasting rounds, for any processor p_i , all processors in G_i exchange at least one beacon with their neighbors in G_i , with high probability. (See the beginning of Section 5.1 for the discussion on the n balls versus a processor p_i for which $|\overrightarrow{G}_i| < n$.)

5.2. 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 showing that the scenario indeed depicts the implementation of the algorithm (which is presented in Fig. 2).

As stated earlier, when we talk about the execution of, or complete iteration of, lines 67 to 80, we do not imply that the branch in lines 75 to 80 necessarily is entered.

Definition 3 (Safe Configurations). Let E be a fair execution of the algorithm presented in Fig. 2 and $c \in E$ a configuration in which $\alpha_i = (\text{leq}(\text{next}_i - 2Du, cT_i) \wedge \text{leq}(cT_i, \text{next}_i))$ 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 of processor p_i . 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 Fig. 2, and c a configuration that is at least u after the starting configuration. Then, it holds that $(\text{leq}(C^i - u, cT_i - w) \wedge \text{leq}(cT_i - w, C^i))$ 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 67, 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 $\text{leq}(t - u, t) \wedge \text{leq}(t, t)$ in c .

Let a'_i be an atomic step that includes the execution of lines 83 to 87 (whether entering the branch or not), 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 $\text{leq}(C^i - u, cT_i - w)$ holds in c' as $\text{leq}(C^i, cT_i - w)$ holds in c . It also follows that $\text{leq}(cT_i - w, C^i)$ holds in c' as $C^i = cT_i - w$ in c . \square

We show that when a processor p_i executes lines 75 to 80 of the algorithm presented in Fig. 2 it reaches a configuration in which α_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 Fig. 2. Moreover, let $a_i \in E$ a step that includes a complete iteration of lines 67 to 80 and c the configuration that immediately follows a_i . Suppose that processor p_i executes lines 75 to 80 in a_i ; then α_i holds in c .

Proof. Among the lines 75 to 80, only lines 78 to 79 can change the values of α_i . Let $t_1 = \text{next}_i$ immediately after line 74 and let $t_2 = \text{next}_i$ immediately after the execution of line 79. We denote by $A = t_2 - t_1$ the value that lines 78 to 79 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 78 and y is the value of cslot_i after line 78. Therefore, $A \in [u, (2D - 1)u]$.

By the claim's assertion, we have that $\text{leq}(cT_i, t_1 + u)$ holds before line 78. Since $u \leq A$, it holds that $\text{leq}(cT_i, t_1 + A)$, and therefore $\text{leq}(cT_i, t_2)$ holds.

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

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

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

Proof. Let p_i be a processor for which α_i does not hold in the starting configuration of E . We show that, within the first complete iteration of lines 67 to 80, the predicate α_i holds. According to Lemma 3, all processors p_i , complete at least one iteration of lines 67 to 80, within a period of u .

Let $a_i \in E$ be the first step in which processor p_i completes the first iteration. If α_i does not hold in the configuration that immediately precedes a_i , then either (1) the predicate in line 68 holds and processor p_i executes line 69 or (2) the predicate of line 74 holds at line 68.

For case (2), immediately after the execution of line 69, the predicate $\neg(\text{leq}(\text{next}_i - 2Du, cT_i) \wedge \text{leq}(cT_i, \text{next}_i))$ does not hold, because $\neg(\text{leq}(t - 2Du, t) \wedge \text{leq}(t, t))$ is false for any t . Moreover, the predicate in line 74 holds, since $\text{leq}(t, t + u)$ holds for any t .

In other words, the predicate in line 74 holds for both cases (1) and (2). Therefore, p_i executes lines 75 to 80 in a_i . By Claim 5, α_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 . \square

We demonstrate the closure property of safe configurations.

Lemma 5. Let E be a fair execution of the algorithm presented in Fig. 2 that starts in a safe configuration c , i.e., a configuration in which α_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 $a_i \in E$ be the first step of processor p_i .

We show that α_i holds in configuration c' that immediately follows a_i . Lines 83 to 87 do not change the value of α_i . By Claim 5, if a_i executes lines 75 to 80 within one complete iteration, then α_i holds in c' . Therefore, we look at step a_i that includes the execution of lines 67 to 74, but does not include the execution of lines 75 to 80.

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 $t_2 - t_1 \bmod T < u$. Furthermore, let $A = \text{next}_i - Du$ and $B = \text{next}_i$ in c . The values of $\text{next}_i - Du$ and $B = \text{next}_i$ do not change in c' . Since α_i is true in c , it holds that $\text{leq}(A, t_1) \wedge \text{leq}(t_1, B)$. We claim that $\text{leq}(A, t_2) \wedge \text{leq}(t_2, B)$. Since $\text{leq}(t_1, B)$ in c , we have that $\text{leq}(t_2, B + t_2 - t_1)$ while p_i executes line 74 in a_i . As a_i does not execute lines 75 to 80, the predicate in line 74 does not hold in a_i . As $\text{leq}(t_1, B)$ and $t_2 - t_1 \bmod T < u$ the predicate in line 74 does not hold iff $\text{leq}(t_2, B)$. Furthermore, we have that $\text{leq}(A, t_1)$, $\text{leq}(t_1, B)$, and $\text{leq}(t_2, B)$. As $0 < t_2 - t_1 \bmod T < u$ we have that $\text{leq}(A, t_2)$. Thus, c' is safe as α_i holds in c' . \square

5.3. Nice executions

We claim that the algorithm (presented in Fig. 2) implements nice executions with high probability. We show that, for any processor p_i , every execution (for which the safe configuration requirements hold) is a nice execution in relation to p_i with high probability.

Theorem 1. Let E be a legal execution of the algorithm presented in Fig. 2. Then, for any processor p_i , E is nice in relation to p_i with high probability.

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 78 and 79 in a_i , a'_i , and a''_i , respectively. The only changes done to next_i from line 79 in a_i to lines 78 and 79 in a'_i are those two lines, which taken together change next_i to $\text{next}_i + Du \bmod T$.

The period of length Du that begins at r and ends at $r' \bmod T$ is divided in D timeslots of length u . A timeslot begins at time $r + xu \bmod T$ and ends at time $r + (x + 1)u \bmod T$ 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 processor p_i , and its set G_i , which includes itself and its neighbors. The processors in \vec{G}_i 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 a processor p_i and R timeframes of length Du . By **Corollary 1**, the probability that all processors $p_j \in G_i$ successfully broadcast at least one beacon to all processors $p_k \in G_i \cap G_j$ is at least $1 - 2^{-\ell}$. Now, let us consider $2R$ timeframes of length Du . Consider the probability that each of the processors $p_j \in G_i$ successfully broadcasts to all processors $p_k \in G_i \cap G_j$ and get a response from all such processors p_k . By **Corollary 1**, that probability is at least $(1 - 2^{-\ell})^2 = 1 - 2^{-\ell+1} + 2^{-2\ell} > 1 - 2^{-\ell+1}$. Therefore, by **Definition 1**, for any processor p_i , E is nice in relation to p_i with high probability. \square

6. Performances of the algorithm

Several elements determine the precision of the clock synchronization. The clock sampling technique is one of them. Elson et al. [9] 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 quality of the approximation of the native clocks of neighboring nodes. Our extensive clock sample records allows for both linear regression and phase-locked looping (see Römer et al. [24]). Moreover, the clock synchronization precision improves as neighboring processors are able to sample each other's clocks more frequently. However, due to the limited energy reserves in sensor networks, careful considerations are required.

Let us consider the continuous operation mode. If the period of the clock samples is too long, the clock precision suffers, as the skews of the native clocks are not constant. Thus, an important measure is $round_i$, where $round_i$ is the time it takes a processor p_i and its neighbors in G_i to exchange beacons and responses. In other words, $round_i$ is the time it takes (1) every processor $p_j \in G_i$ (including p_i) to successfully broadcast at least one beacon to all processors $p_k \in G_i \cap G_j$ and (2) every such processor p_j to get at least one response from all such processors p_k .

Let us consider ideal system settings in which broadcasts never collide. In the worst case, $|G_i| = |\vec{G}_i| = n$. Sending n beacons and getting n responses to each of these beacons requires the communication of at least $O(n^2)$ samples. 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 up to n processors. Hence, $u \in O(n \log n)$. Therefore $round_i \in O(n^2(\log n)^2)$. Moreover, with a probability of at least $1 - 2^{-\ell+1}$, the algorithm can secure a clock sampling period that is $O((\log n)^2)$ times the optimum.

We note that the required storage is in $O(n^2 \log n \log T)$. By **Lemma 4**, starting in an arbitrary configuration, our system stabilizes within u time, and as we have seen above $u \in O(n \log n)$.

6.1. Optimizations

We can use the following optimization, which is part of many existing implementations. Before accessing the communication media, a processor 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, from a processor $p_j \in G_i$, that it started receiving (and did not finish) before time $t - d$, where t is the time of the broadcast by p_i . In that case it aborts its message. For p_i , and for the sake of the worst-case analysis, this counts as a collision. However, for p_j it is a successful broadcast (assuming that the message is not lost to noise or to collision with another message).

7. 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 and 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., [6,8,7]), which consider multi-channel radio networks). We note that many of the existing implementations assume the existence of a fine-grained synchronized clock, which we implement.

Message scheduling is important for clock synchronization. Moradi et al. [22] compare clock synchronization algorithms for wireless sensor networks considering precision, cost and fault tolerance. They show that, without a message scheduling algorithm of some sort, the Reference Broadcast algorithm of [9] suffers heavily from collisions.

Ganeriwat et al. [12] overcome the challenge of delayed beacons using the round-trip synchronization technique. With this technique the average delay of a message from processor p_i to processor $p_j \in G_i$, and a message back from p_j to p_i , can be calculated using the send and receive times of those messages. Thus, a delay attack can be detected if the delay is larger than some known upper bound on message delay. They use the Byzantine agreement protocol [17] for a cluster of g nodes where all g nodes are within transmission range of each other. Thus, Ganeriwat et al. require $3f + 1 \leq g$. Song et al. [28] 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 $2f + \epsilon \leq g$, where g is the

number of neighbors and where ϵ is a safety constant (see [28]). We note that both approaches are applicable to our work. We further note that a processor $p_k \in G_j \cap G_i$ can detect delay attacks against beacons that nodes p_i and p_j have sent to each other, by the mechanisms of calculating average message delay and comparing with a known upper bound. This is possible because p_k gets send and receive times of messages back and forth between p_i and p_j .

Based on our practical assumptions, we are able to avoid the Byzantine agreement overheads and follow the approach of Song et al. [28]. We can construct a self-stabilizing version of their strategy, by using our sampling algorithm and by detecting outliers using the generalized extreme studentized deviate (GESD) algorithm [25]. Let B be the set of delivered beacon records within a period of \mathcal{R} and test the set B for outliers using the GESD algorithm.

Existing implementations of secure clock synchronization protocols [31,30,11,10,20,12,28] 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. [29] describe a cluster-wise synchronization algorithm that is based on synchronous broadcasting rounds. The authors assume that a Byzantine agreement algorithm [17] 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. [20] describe several possible attacks on an (unsecured) clock synchronization algorithm and suggest countermeasures. 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 [10] 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. [30] investigate how to use multiple clocks from external source nodes (e.g., base stations) to increase the resilience against an attack that captures source nodes. In this work, there are no source nodes.

In [31], 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 [14] 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.

7.1. 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 use system settings that consider many practical issues, 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 reinforcing 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 f captured and/or pulse-delay attacked nodes in a neighborhood at all times, for a setting where $3f + 1 \leq n$ must hold in the neighborhood. Since sensor networks reside in an unattended environment, the last assumption is unrealistic when considering long timespans.

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.

Acknowledgements

This work would not have been possible without the contribution of Marina Papatrifiantilou 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] Joffroy Beauquier, Synnöve Kekkonen-Moneta, Fault tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors, *International Journal of Systems Science* 28 (11) (1997) 1177–1187.
- [3] Murat Demirbas, Anish Arora, Tina Nolte, Nancy A. Lynch, A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks, in: OPODIS, in: LNCS, vol. 3544, Springer, 2004, pp. 299–315.
- [4] Edsger W. Dijkstra, Self-stabilizing systems in spite of distributed control, *Communications of the ACM* 17 (11) (1974) 643–644.
- [5] Shlomi Dolev, *Self-Stabilization*, MIT Press, 2000.
- [6] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, Fabian Kuhn, Calvin C. Newport, The wireless synchronization problem, in: PODC, ACM, 2009, pp. 190–199.
- [7] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, Calvin C. Newport, Gossiping in a multi-channel radio network, in: DISC, in: *Lecture Notes in Computer Science*, vol. 4731, Springer, 2007, pp. 208–222.
- [8] Shlomi Dolev, Seth Gilbert, Rachid Guerraoui, Calvin C. Newport, Secure communication over radio channels, in: PODC, ACM, 2008, pp. 105–114.
- [9] Jeremy Elson, Lewis Girod, Deborah Estrin, Fine-grained network time synchronization using reference broadcasts, *Operating Systems Review (ACM SIGOPS)* 36 (SI) (2002) 147–163.
- [10] Emerson Farrugia, Robert Simon, An efficient and secure protocol for sensor network time synchronization, *Journal of Systems and Software* 79 (2) (2006) 147–162.
- [11] Saurabh Ganeriwal, Srdjan Capkun, Chih-Chieh Han, Mani B. Srivastava, Secure time synchronization service for sensor networks, in: *Proceedings of the 4th ACM workshop on Wireless security, WiSe'05*, ACM Press, NYC, NY, USA, 2005, pp. 97–106.
- [12] Saurabh Ganeriwal, Srdjan Capkun, Mani B. Srivastava, Secure time synchronization in sensor networks, *ACM Transactions on Information and Systems Security* (2008).
- [13] Seth Gilbert, Rachid Guerraoui, Calvin C. Newport, Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks, in: OPODIS, in: LNCS, vol. 4305, Springer, 2006, pp. 215–229.
- [14] Ted Herman, Chen Zhang, Best paper: Stabilizing clock synchronization for wireless sensor networks, in: SSS, in: LNCS, vol. 4280, Springer, 2006, pp. 335–349.
- [15] Richard Karp, Jeremy Elson, Deborah Estrin, Scott Shenker, Optimal and global time synchronization in sensor networks, Technical report, Center for Embedded Networked Sensing, Univ. of California, Los Angeles, 2003.
- [16] Richard M. Karp, Jeremy Elson, Christos H. Papadimitriou, Scott Shenker, Global synchronization in sensor networks, in: LATIN, in: LNCS, vol. 2976, Springer, 2004, pp. 609–624.
- [17] Leslie Lamport, Robert E. Shostak, Marshall C. Pease, The byzantine generals problem, *ACM Transactions on Programming Languages and Systems* 4 (3) (1982) 382–401.
- [18] Christoph Lenzen, Thomas Locher, Roger Wattenhofer, Clock synchronization with bounded global and local skew, in: 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS, Philadelphia, Pennsylvania, USA, October 2008, pp. 509–518.
- [19] Christoph Lenzen, Thomas Locher, Roger Wattenhofer, Tight bounds for clock synchronization, in: 28th ACM Symposium on Principles of Distributed Computing, PODC, Calgary, Canada, August 2009, pp. 46–55.
- [20] Michael Manzo, Tanya Roosta, 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*, ACM Press, NYC, NY, USA, 2005, pp. 107–116.
- [21] Michael Mitzenmacher, Eli Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, New York, NY, USA, 2005.
- [22] Farnaz Moradi, Asrin Javaheri, Clock synchronization in sensor networks for civil security, Technical report, Computer Science and Engineering, Chalmers University of technology, March 2009.
- [23] Kay Römer, Time synchronization in ad hoc networks, in: *MobiHoc '01: Proceedings of the 2nd ACM International Symposium on Mobile ad hoc Networking & Computing*, ACM Press, NYC, NY, USA, 2001, pp. 173–182.
- [24] Kay Römer, Philipp Blum, Lennart Meier, Time synchronization and calibration in wireless sensor networks, in: *Handbook of Sensor Networks: Algorithms and Architectures*, John Wiley and Sons, 2005, pp. 199–237.
- [25] B. Rosner, Percentage points for a generalized *esd* many-outlier procedure, *Technometrics* 25 (1983) 165–172.
- [26] Bruce Schneier, *Applied Cryptography*, 2nd edition, John Wiley & Sons, 1996.
- [27] Philipp Sommer, Roger Wattenhofer, Gradient clock synchronization in wireless sensor networks, in: 8th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN, San Francisco, USA, April 2009.
- [28] Hui Song, Sencun Zhu, Guohong Cao, Attack-resilient time synchronization for wireless sensor networks, *Ad Hoc Networks* 5 (1) (2007) 112–125.
- [29] Kun Sun, Peng Ning, Cliff Wang, Fault-tolerant cluster-wise clock synchronization for wireless sensor networks, *IEEE Transactions on Dependable and Secure Computing* 2 (3) (2005) 177–189.
- [30] Kun Sun, Peng Ning, Cliff Wang, Secure and resilient clock synchronization in wireless sensor networks, *IEEE Journal on Selected Areas in Communications* 24 (2) (2006) 395–408.
- [31] Kun Sun, Peng Ning, Cliff Wang, Tinsysersync: secure and resilient time synchronization in wireless sensor networks, in: *ACM Conference on Computer and Communications Security, ACM, 2006*, pp. 264–277.