Secure and Self-stabilizing Clock Synchronization in Sensor Networks

Jaap-Henk Hoepman[1], Andreas Larsson[2], Elad M. Schiller[2], and Philippas Tsigas[2]

[1] TNO ICT, and Radboud University Nijmegen
jaap-henk.hoepman@tno.nl
[2] Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University
{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.

Keywords: 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 which 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

T. Masuzawa and S. Tixeuil (Eds.): SSS 2007, LNCS 4838, pp. 340–356, 2007.

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 propagation delay of messages in short distance 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 [9,10,19]).

We are interested in fine-grained clock synchronization, where there are no cryptographic counter measures for such pulse-delay attacks, e.g., 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 [18]). Unfortunately, the lack of fine-grained clock synchronization implies that the round-trip time of message exchange cannot be efficiently estimated.

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., [18,22]). 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 a model that is comparable to the one of Gilbert et al. [11], which considers 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, which 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] uses Byzantine agreement (requires $3t + 1 \leq n$), and [19] considers the statistical outliers (requires $2t + \epsilon \leq n$, where $\epsilon \in O(1)$).

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 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 $O((\log n)^3)$ times the optimum, where $\Omega(n^2)$ is the optimum and 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.

Document structure. We start by describing the system settings (Section 2) and formally present the algorithm (Section 3). Then we review the literature and draw our conclusions (Section 4).

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.

Time, Clocks, and Their Notation. We follow a model compatible with the one 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. 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. The relative clock skew is $\rho_{i,j} = \rho_i - \rho_j$. We assume that $1 - \kappa \leq \rho_i \leq 1 + \kappa$. The second derivative of the clocks' offset is called *drift*. We follow the approach of Herman and Zhang [12] and allow non-zero drift (as long as $\rho_i \in [\rho_{\min}, \rho_{\max}]$).

Communications. Wireless transmissions are subject to collision and noise. The processors communicate among themselves using a local broadcast primitive, *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 others 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 (with whose communications, processor p_i can interfere) by $G_i \subseteq \{p_j \in P | R_{lb} \geq |L_i - L_j|\}$ (respectively, $\overrightarrow{G_i} \subseteq \{p_j \in P | 2R_{lb} \geq |L_i - L_j|\}$). We assume that $n \geq |\overrightarrow{G_i}|$ is a known upper bound on the node's degree.

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 in which m enters the communication channel and the time in 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.

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 have started (and did not finish) before time $t - d$, where t is the time of the broadcast by p_i.

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., [18,22]). 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 suitable for fine-grained clock synchronization in the presence of pulse-delay attacks.

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 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 time) slot*, where w is the time it takes to execute a complete iteration of the do-forever loop, including all messages received in that slot, assuming that there is a known upper bound on the number of those. Since processors execute programs other than the clock synchronization, the actual time in which the timer goes off is hard to predict. Therefore, for the sake of simplicity, we assume that the this time has a uniform distribution. We note that a simple random scheduler can be used for the case in which the this time can be characterized.

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* $E = c[0], a[0], c[1], a[1], \ldots$ 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.

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 [22]).

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 (approximately) t. We note that every step can be associated with at most one communication operation and therefore with one access to the native clock counter 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.

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 omni-directional 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 receive either no beacon or beacons after a delay that is greater than a known constant.

Concurrent vs. Independent Broadcasts. We say that processor p_i performs an *independent broadcast* in a step $a_i \in E$ if there is no processor $p_j \in P$ 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 $p_k \in P$), 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. 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.

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 β broadcasts of the algorithm, where β 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 α messages the adversary can intercept at most β. In other words, our assumption regarding the ratio of β/α is comparable to the model of Gilbert et al. [11]. The parameter $\xi \geq 1$ denotes the maximal number of repeated transmissions required for a single successful message transfer whenever there are no message collisions due to the algorithm's concurrent broadcasts. We assume that all processors know ξ.

We say that execution E has *fair communications*, if, whenever processor p_i independently broadcasts ξ successive messages in steps $a_i^{\xi} \in E$, 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.

The Environment. The environment that restricts the adversary's ability to launch message interception attacks guarantees fair communication. The environment can execute the operation $\mathit{omission}_i(m)$ (which is associated with a particular message, m, sent by processor p_i) immediately after $\mathit{LBcast}_i(m)$. 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 algorithm 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.

System Specifications ***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).

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.

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. [16] 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 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 neighbouring clocks. We measure the algorithm's performance by looking at the period, $\Gamma(n)$, it takes n processors to send at least one beacon that all processors respond to. In other words, we are interested in the minimal period in which all processors are able to complete roundtrip message exchange.

Let p_i, p_j, and p_k be three correct nodes such that p_i and p_j are of type (1) and p_k is of type (2). 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_k detects that p_j 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 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 clocks after neutralizing the effect of pulse-delay attacks (see section 4 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 responses. 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 response. 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 there are no concurrent broadcasts. Our scheduling strategy is simple; the processors choose a random time to broadcast from a predefined period D. We use time redundancy to overcome the clocks' asynchrony and the difficulty in measuring D. Moreover, we use a parameter, ℓ, used to trade off between minimal size of D and the probability of having a collision free schedule.

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. (We note that $BLog$ may depend on the safety parameter, ℓ, for assuring that all nodes successfully broadcast.) 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 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 $\langle t_s^j(k), t_r^j(k) \rangle$, 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.

The Algorithm's Pseudo-code. The pseudo-code, in Figure 1, includes two procedures: (1) a do-forever loop that schedules and broadcasts beacon messages (lines 53 to 63) and (2) an upon message arrival procedure (lines 66 to 68).

The Do-Forever Loop. Recall that by our system settings assumptions (Section 2), we assume that 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 do-forever loop periodically tests whether the "timer" has expired (in lines 53 to 58). In case the beacon's next schedule is in the "too far in the past" or "too far in the future", then processor p_i "forces" the "timer" to expire (line 55). The algorithm tests that all the stored beacon messages are ordered correctly and refer to the last $BLog$ beacons (line 56). In the case where the stored beacon messages are incorrect, then the algorithm flushes the queues (line 57).

When the time slot arrives, the processor outputs a synchronizer case record, a response to the beacons that processor p_i has sent $BLog$ rounds ago (line59). 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 60). The next schedule for processor p_i is set (lines 61 and 62) just before it broadcasts the beacon message (line 63).

The Message Arrival. When a beacon message arrives (line 65), processor p_i outputs a record of the non-synchronizer case (line 68). This is not done before

```
Constants:
2  i = id of executing processor
   n = total number of processors
4  w = compensation time between lines 53 and 63
   d = upper bound on message propagation delay
6  u = size of a slot in time units (u > d + w)
   BLog = ⌈(ξ + 2)(ρmax − ρmin)⌉, backlog size
8  ℓ = the safety parameter
   D = ℓ n log n, the broadcast time slots
10 T = number of possible states of a timestamp (T ≫ Du)

12 Variables:
   native_clock : immutable storage of the native clock
14 m[n] = all received messages and timesamps
     each entry is an array v[n]
16     each entry is a queue q[BLog]
         each entry is a pair ⟨s, r⟩
18 cslot : [0, D-1] = current slot in use
   next : [0, T -1] = schedule of next broadcast
20 cT = last do-forever loop's timestamp

22 External functions:
   output(R) : delivers record R to the upper layer
24 choose(S) : uniform selection of an item from the set S
   siz(Q) : size of the queue
26 fst(Q) : least recently enqueued element in Q, number 0
   lst(Q) : most recently enqueued element in Q
28 flush(Q) : empties the queue Q
   get(t,Q) : list elements of field t ∈ {s,r} in Q
30
   Macros and inlines:
32 border(t) : (D-cslot)u + t   mod T
   schedule(t) : cslot·u + t   mod T
34 leq(x, y) : (∃ b : 0 ≤ b ≤ 2 BLog D u ∧
                  y  mod T = x + b  mod T)
36 enq(q, m) : {while full(q) do dequeue(q); enqueue(m) }
   cvec(v,t) : siz(v) = 0 ∨ (leq(fst(v),t) ∧ leq(lst(v),t) ∧
38   {∀ b1 < b2, {b1,b2} ⊆ [1,siz(v)] : leq(v[b1],v[b2])}
   checki(t) : cvec(get(s,m[i].v[i].q),t)
40 check(t) : ∧ {∀ j ∈ P-{i} : cvec(get(r,m[i].v[j].q),t)}
   (* Get response-record for pj, for pi as the synchronizer *)
42 tsi(s, j) : {if ∄ b : s = m[j].v[i].q[b].s then return ⊥
     else return
44     ⟨m[j].v[i].q[b].r, lst(m[i].v[j]).s, lst(m[i].v[j]).r⟩ }
   matches(j) : {b : tsi(m[i].v[i].q[b].s, j) ≠ ⊥}
46 sci(j) : if matches(j) = ∅ then return ⊥
     else return min(matches(j))
48 (* Get response-record for pk, for pj as the synchronizer *)
   ts(s, j, k) : details appear in [14].
50 sc(j, k) : details appear in [14].

52 Do forever, every u/2
   let cT = read(native_clock) + w
54 if ¬ (leq(next-2Du, cT) ∧ leq(cT, next+u)) then
     next ← cT
56 if ¬ (checki(cT) ∧ check(cT)) then
     ∀ j,k ∈ P : flush(m[j].v[k].q)
58 if leq(next, cT) ∧ leq(cT, next + u) then
     output ⟨i, { ⟨sci(j), j, tsi(sci(j), j)⟩ : j ∈ P -{i}} ⟩
60   enq(m[i].v[i].q, ⟨cT, ⊥⟩)
     (next, cslot) ← (border(next), choose([0, D-1]))
62   next ← schedule(next)
     LBcast(m[i])
64
   Upon LBrecv(j, r, v)        (* i ≠ j *)
66 enq(m[i].v[j].q, ⟨lst(v[j].q).s, r⟩)
   m[j] ← v
68 output ⟨j, {⟨sc(j,k), k, ts(sc(j, k), j, k)⟩ : k ∈ P -{i, j}}⟩
```

Fig. 1. Secure and self-stabilizing native clock sampling algorithm (code for $p_i \in P$)

processor p_i stores the arrival time of the message (line 66) and the message itself (line 67). These data can be used for the reference broadcast in the upper layer. Once p_i receives a beacon from node p_k, node p_i scans $m[\,]$ for responses that refer to p_k's previous beacons.

The Correctness. We divide the correctness proof of the algorithm presented in Figure 1 into two parts. The first part relates to the task of random broadcast scheduling and the second relates to the task of beacon and response aggregation. The second part's proofs simply verify that the pseudo-code aggregates the right responses with the right beacon. Due to space limits, some parts of the correctness proof of the random broadcast scheduling, and the correctness proof of the aggregation task appears in [14].

We analyze our random broadcasting strategy as a ball throwing game in a team of n players that throw balls into bins. The bins represent the timeslots. For the sake of simplicity, we consider every timestamp as a single information unit, which we call a ball. The players' coordination is poor and resembles the clocks' partial synchrony. We measure the team performance by looking at the

number, $\Gamma(n)$, of bins it takes the team to get each at least n balls into bins. A detailed game description and the correctness proof of corollary 1 appears in [14].

Corollary 1. *$\Gamma(n) \in \Omega(n^2)$ and the random broadcasting strategy of the algorithm presented in Figure 1 secures with probability $1 - 2^{-\ell}$ that $\Gamma(n) \in O(n^2(\log n)^3)$.*

Let E be an execution and $a_i \in E$ an atomic step in which processor p_i broadcasts. Let $c \in E$, be the configuration that immediately follows a_i. We define the first *round* from a_i, $E_{a_i}(1)$ as a (finite) subsequence of E that starts in c and ends in the atomic step $a_i' \in E$, that is the first step after a_i in which p_i broadcasts. We define the second round $E_{a_i}(2) = E_{a_i'}(1)$. Similarly, the *x-th* round $E_{a_i}(x)$, $\forall x > 1, x \in \mathbb{N}$, is defined as $E_{a_i}(x) = E_{a_i'}(x-1)$. We say that processor p_j *skips* a round $E_{a_i}(1)$ if p_i does not receive a broadcast from p_j in $E_{a_i}(1)$. The *beacon broadcast period* (BBP) of processor p_i for a given broadcast in atomic step $a_i \in E$ is the real time length of the round $E_{a_i}(1)$.

Definition 1. *We define the set LE_{rbs} of legal executions with respect to the task of random broadcast scheduling, such that it includes every execution E in which: (1) The expected* beacon broadcast period *(BBP) of processor p_i is within $[Du/\rho_{\max}, Du/\rho_{\min}]$ and (2) The probability that no processor skips ξ consecutive rounds $E_{a_i}(x), \ldots E_{a_i}(x+\xi)$ is in $O(1-2^{-\ell})$, where $x \in \mathbb{N}$ and $a_i \in E$, is an atomic step in which p_i broadcasts).*

Let E be a fair execution of the algorithm presented in Figure 1 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. We say that c is safe with respect to LE_{rbs}.

We show that cT_i follows the native clock.

Lemma 1. *Let E be a fair execution of the algorithm presented in Figure 1, 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 53, 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 66 to 68, 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 $t' - t < u/2$. ∎

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

Lemma 2. *Let E be a fair execution of the algorithm presented in Figure 1. 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 53 to 63, the predicate α_i holds. According to Lemma 1, all processors, p_i, complete at least one iteration of lines 53 to 63, 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 the predicate in line 54 holds and processor p_i executes line 55.

Immediately after the execution of line 55, 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 58 holds, since $\mathit{leq}(t, t+u)$ holds for any t. Therefore, p_i executes lines 59 to 63.

Claim. Suppose that the predicate $\neg(\mathit{leq}(next_i-2Du, cT_i)\wedge\mathit{leq}(cT_i, next_i))$ (line 54) does not hold and the predicate $\mathit{leq}(next_i, cT) \wedge \mathit{leq}(cT, next_i + u)$ (line 58) holds. If processor p_i executes lines 59 to 63, then α_i holds for the configuration that immediately follows.

Proof. Among the lines 59 to 63, only lines 61 to 62 can change the values of α_i. Let $t_1 = next_i$ immediately after line 58 and let $t_2 = next_i$ immediately after the execution of line 62. We denote by $A = t_2 - t_1$ the value that lines 61 to 62 adds 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 61 and y is the value of $cslot_i$ after line 61.

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 61. 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 a safe configuration follows the configuration of Definition 1.

Lemma 3. *Let E be a fair execution of the algorithm presented in Figure 1 that starts in a configuration c, in which α_i holds. Then, every configuration in E is safe with respect to LE_{rbs}.*

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. According to Lemma 1 and by the fairness of E, without loss of generality, we can assume that $C^i - t_i \bmod T \leq u/2$.

We show that α_i holds in configuration c' that immediately follows a_i. Lines 66 to 68 do not change the value of α_i. By the proof of Lemma 2, if a_i executes lines 59 to 63 within one complete iteration, then α_i holds in c'. Therefore, we look at step a_i that includes the execution of line 53 to 58, but does not include the execution of lines 59 to 63.

Let $t_1 = cT_i$ in c and $t_2 = cT_i$ in c'. We show that while a_i executes line 54, the predicate $\neg(\mathit{leq}(next_i - 2Du, cT_i) \wedge \mathit{leq}(cT_i, next_i))$ does not hold in a_i.

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 α_i is true in c, it holds that $\mathit{leq}(A, t_1) \wedge \mathit{leq}(t_1, B)$. We claim that $\mathit{leq}(A, t_2) \wedge \mathit{leq}(t_2, B+u)$. Suppose, in a way of contradiction, that $\mathit{leq}(A, t_2) = \mathit{leq}(A, t_1 + u/2)$ does not hold. Then, $\mathit{leq}(next_i - Du, t_1 + u/2)$ does not hold in configuration c, which implies that $\mathit{leq}(t_1 - Du, t_1 + u/2)$ because $\mathit{leq}(t_1, next_i)$ hold in c. Hence, a contradiction.

Since $\mathit{leq}(t_1, B)$ in c, we have that $\mathit{leq}(t_2, B+u)$ while p_i execute line 54 in a_i. By the assumption that $t2 - t1 \bmod T < u$ we have that $\mathit{leq}(t_1 + u/2, B+u) \implies \mathit{leq}(t_2, B + u)$. ■

We show that every execution (for which the safe configuration requirements hold) is a legal execution with regard to the random broadcast scheduling task.

Lemma 4. *Let E be a fair execution of the algorithm presented in Figure 1, where all configurations in E are safe. Then, $E \in LE_{rbs}$.*

Proof. **(1)** Let $a_i \in E$ be a step in which processor p_i broadcasts and $a_i' \in E$ is the first step after a_i in which p_i broadcasts. Let $c_1 \in E$ immediately precede a_i and $c_2 \in E$ immediately follow a_i. Let $c_3 \in E$ immediately precede a_i' and $c_4 \in E$ immediately follow a_i'. Let $n_1 = next_i$ in c_1, $t_1 = cT_i$ in c_2, $n_2 = next_i$ in c_3, $t_2 = cT_i$ in c_4. The BBP can be expressed as B/ρ_i, where $B = t_2 - t_1 \bmod T$.

Processor p_i broadcasts in line 63 only when the predicate $\gamma_i = \mathit{leq}(next_i, cT_i) \wedge \mathit{leq}(cT_i, next_i + u)$ (line 58) holds. Claim 3 of Lemma 2 shows that in lines 61 to 62, $next_i$ is incremented (modulo T) by $A = (y + D - x)u$. Both integers x and y are chosen independently and from the same uniform distribution over $[0, D-1]$. Therefore, they have the same expected value. Therefore, the expected value of A is $\mathbf{E}((y + D - x)u) = (\mathbf{E}(y) + \mathbf{E}(D) - \mathbf{E}(x))u = (\mathbf{E}(D) + \mathbf{E}(y) - \mathbf{E}(y))u = \mathbf{E}(D)u = Du$.

Let $\hat{u}_1 = t_1 - n_1$ and $\hat{u}_2 = t_2 - n_2$. If we assume that $\hat{u}_1$ and $\hat{u}_2$ are independent and have the same distribution, the expected value of B is $\mathbf{E}((n_2 + \hat{u}_2) - (n_1 + \hat{u}_1)) = \mathbf{E}(n_2 - n_1) + \mathbf{E}(\hat{u}_2 - \hat{u}_1) = Du + \mathbf{E}(\hat{u}_2) - \mathbf{E}(\hat{u}_1) = Du$

Even if $\hat{u}_1$ and $\hat{u}_2$ are not independent and/or not from the same distribution, the expected value of B is Du as well, as the decrement of the BBP for a broadcast in a_i' within the period $[n_2, n_2 + u]$ implies a corresponding increment of the BBP for the broadcast in a_i. By the definition of $\rho_{\min}$ and $\rho_{\max}$ we have that $Du/\rho_{\max} \leq Du/\rho_i \leq Du/\rho_{\min}$ (since $\forall i : \rho_{\min} \leq \rho_i \leq \rho_{\max}$).

(2) Let a_i be a step in which in which processor p_i broadcasts, a_i' be the next step in which processor p_i broadcasts, and c be the configuration that immediately follows a_i.

Let r be the value of $next_i$ between lines 61 and 62 in a_i. The period of length Du that begins at r is divided in D slots of length u. A slot begins at time $r + xu$ and ends at time $r + (x+1)u$ for a unique integer $x \in [0, D-1]$. The slot in which a_i' broadcasts is *cslot* in c. By Corollary 1, the probability of no messages collides in the period r to $r + Du$ is in $O(1 - 2^{-\ell})$. ■

Performances. 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 roundtrip 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. [16]). 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.

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.

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 to respond to all n beacons. By Corollary 1, $round(n) \in \Omega(n^2)$ and $round(n) \in O(n^2(\log n)^3)$. Therefore, our design can secure clock precision that is $O((\log n)^3)$ times the optimum, with probability that is $1 - 2^{-\ell}$.

We note that the required storage is in $O(n^2 \log T)$. Moreover, existing sensor networks technology allows a message size of $14n+O(1)$ bytes. In [14], we explain how to further accommodate message size and to optimize performance.

4 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. requires $3t + 1 \leq n$. Song et al.'s [19] 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 stabile for $2t + \epsilon \leq n \leq 3t + 1$, where ϵ is a safety constant (see [19]). 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. [19]. They assume the existence of a distributed algorithm for sending beacons and collecting their responses. This work presents the first design of that algorithm.

The generalized extreme studentized deviate (GESD) algorithm [17] can be used to detect outliers. We note that there exists self-stabilizing version of Song et al.'s [19] strategy. Let B be the set of "recently" delivered beacon records. By "recently", we mean that within a predefined period, $\varrho \in O(D)$, the node removes old records from B, where ϱ depends on ξ, i.e., the number of broadcasts it takes to assure message delivery. The algorithm tests set B for outliers.

Existing implementations of secure clock synchronization protocols [22,21,9,8,15,10,19] 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. [20] describe a cluster-wise synchronization algorithm that is based on synchronous 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 rounds or start.

Manzo et al. [15] 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. [21] 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 [22], 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.

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, e.g., the nonce techniques cannot resist pulse-delay attacks.

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 follow the original designer's assumptions. Interestingly, the self-stabilization design criteria provide an elegant way for designing secure autonomous systems.

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

References

1. Abramson, N., et al.: The Aloha System. Univ. of Hawaii (1972)
2. Demirbas, M., Arora, A., Nolte, T., Lynch, N.A.: A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 299–315. Springer, Heidelberg (2005)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)

5. Dolev, S., Gilbert, S., Guerraoui, R., Newport, C.: Gossiping in a Multi-Channel Radio Network, An Oblivious Approach to Coping with Malicious Interference. In: DISC 2007. LNCS, vol. 4731, pp. 130–145. Springer, Heidelberg (2007)
6. Elson, J., Girod, L., Estrin, D.: Fine-grained network time synchronization using reference broadcasts. Operating Systems Review (ACM SIGOPS) 36(SI), 147–163 (2002)
7. Elson, J., Karp, R.M., Papadimitriou, C.H., Shenker, S.: Global synchronization in sensornets. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 609–624. Springer, Heidelberg (2004)
8. Farrugia, E., Simon, R.: An efficient and secure protocol for sensor network time synchronization. J. Syst. Softw. 79(2), 147–162 (2006)
9. Ganeriwal, S., Capkun, S., Han, C.-C., Srivastava, M.B.: Secure time synchronization service for sensor networks. In: Ngu, A.H.H., Kitsuregawa, M., Neuhold, E.J., Chung, J.Y., Sheng, Q.Z. (eds.) WISE 2005. LNCS, vol. 3806, pp. 97–106. Springer, Heidelberg (2005)
10. Ganeriwal, S., Capkun, S., Srivastava, M.B.: Secure time synchronization in sensor networks. ACM Transactions on Information and Systems Security (March 2006)
11. Gilbert, S., Guerraoui, R., Newport, C.C.: Of malicious motes and suspicious sensors: On the efficiency of malicious interference in wireless networks. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 215–229. Springer, Heidelberg (2006)
12. Herman, T., Zhang, C.: Best paper: Stabilizing clock synchronization for wireless sensor networks. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 335–349. Springer, Heidelberg (2006)
13. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
14. Larsson, A., Schiller, E.M., Tsigas, P.: Secure and fault-tolerant clock synchronization in sensor networks. TR 2006:16, Computer Science and Engineering, Chalmers University of technology (September 2006)
15. Manzo, M., Roosta, T., Sastry, S.: Time synchronization attacks in sensor networks. In: SASN 2005. Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks, pp. 107–116. ACM Press, New York (2005)
16. Römer, K., Blum, P., Meier, L.: Time synchronization and calibration in wireless sensor networks. In: Stojmenovic, I. (ed.) Handbook of Sensor Networks: Algorithms and Architectures, pp. 199–237. John Wiley and Sons, Chichester (2005)
17. Rosner, B.: Percentage points for a generalized *esd* many-outlier procedure. Technometrics 25, 165–172 (1983)
18. Schneier, B.: Applied Cryptography, 2nd edn. John Wiley & Sons, Chichester (1996)
19. Song, H., Zhu, S., Cao, G.: Attack-resilient time synchronization for wireless sensor networks. Ad Hoc Networks 5(1), 112–125 (2007)
20. Sun, K., Ning, P., Wang, C.: Fault-tolerant cluster-wise clock synchronization for wireless sensor networks. IEEE Transactions on Dependable and Secure Computing 2(3), 177–189 (2005)
21. Sun, K., Ning, P., Wang, C.: Secure and resilient clock synchronization in wireless sensor networks. IEEE Journal on Selected Areas in Communications 24(2), 395–408 (2006)
22. Sun, K., Ning, P., Wang, C.: Tinysersync: secure and resilient time synchronization in wireless sensor networks. In: Juels, A., Wright, R.N., di Vimercati, S.D.C. (eds.) ACM Conference on Computer and Communications Security, pp. 264–277. ACM Press, New York (2006)