# Gradient Clock Synchronization in Wireless Sensor Networks

Philipp Sommer
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
sommer@tik.ee.ethz.ch

Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

## ABSTRACT

Accurately synchronized clocks are crucial for many applications in sensor networks. Existing time synchronization algorithms provide on average good synchronization between arbitrary nodes, however, as we show in this paper, *close-by nodes* in a network may be synchronized poorly. We propose the Gradient Time Synchronization Protocol (GTSP) which is designed to provide accurately synchronized clocks between neighbors. GTSP works in a completely decentralized fashion: Every node periodically broadcasts its time information. Synchronization messages received from direct neighbors are used to calibrate the logical clock. The algorithm requires neither a tree topology nor a reference node, which makes it robust against link and node failures. The protocol is implemented on the Mica2 platform using TinyOS. We present an evaluation of GTSP on a 20-node testbed setup and simulations on larger network topologies.

## Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

## General Terms

Algorithms, Measurements

## Keywords

Sensor Networks, Time Synchronization, Clock Drift, Implementation, Experiments

## 1. INTRODUCTION

A wireless sensor network is a promising novel tool for observing natural phenomena at large scale or high resolution. Without doubt, *time* is a first-class citizen in wireless sensor networks. Without accurate time (and similarly location) information, sensed data often loses valuable context. Although one can imagine applications where the "when and where" of the sensed data is of no great concern, a majority of applications will prefer to tag the measured data with a timestamp. Such a timestamp will only be meaningful if the nodes in the wireless sensor network manage to have an adequate agreement of time. Indeed, there are sensor networks that can estimate the location of an event, simply by using trilateration on an acoustic signal [22, 2].

In addition, time synchronization is significant as sensor network protocols make use of time in various forms. Media access control using TDMA needs accurate time information, so that transmissions do not interfere. Similarly, to save energy, sensor network protocols often employ advanced duty-cycling schemes, and turn off their radio if not needed [3]. An accurate time helps to save energy by shortening the necessary wake-up guard times.

Although each sensor node is equipped with a hardware clock, these hardware clocks can usually not be used directly, as they suffer from severe drift. No matter how well these hardware clocks will be calibrated at deployment, the clocks will ultimately exhibit a large skew. To allow for an accurate common time, nodes need to exchange messages from time to time, constantly adjusting their clock values.

Although multi-hop clock synchronization has been studied extensively in the last decade, we believe that there are still facets which are not understood well, and eventually need to be addressed. One such issue is *locality*: Naturally, one objective in clock synchronization is to minimize the skew between any two nodes in the network, regardless of the "distance" between them. This is known as *global* clock skew minimization. Indis-

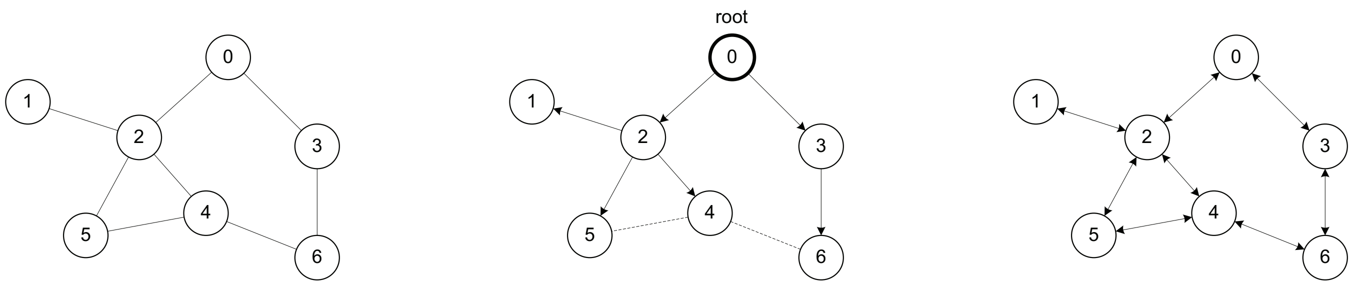


**Figure 1: On the left we see a typical sensor network; edges between sensor nodes indicate a bidirectional communication link. The center figure represents a tree-based synchronization protocol with node 0 as reference clock (root), where every node in the tree synchronizes with its parent; in this example we would expect nodes 4 and 6 to synchronize suboptimally even though they are direct neighbors, because they are part of different subtrees. Finally, on the right we see the idea of a gradient synchronization protocol: Every node synchronizes with all its neighbors in the communication graph. No root node is necessary.**

putable, having two far-away nodes well-synchronized is a noble goal, but is it really what we require most? In this paper, we argue that accurate clock synchronization between neighboring nodes is often at least as important. In fact, all examples mentioned earlier tolerate a suboptimal global clock synchronization: Guessing the location of a commonly sensed acoustic signal needs a precise clock synchronization between all the nodes that are able to sense the signal. Similarly, in a MAC layer that is optimized for throughput or energy, we care that possibly interfering (neighboring) nodes have a precise clock. In contrast, global skew is not of great concern, it is perfectly tolerable if far-away nodes have larger pair-wise error. This is known as *local* clock skew minimization. Optimally, we would like to have a clock synchronization protocol that is precise in the direct neighborhood, and maybe a bit less so in the extended neighborhood.

Current state-of-the-art multi-hop clock synchronization protocols such as FTSP [14] are designed to optimize the global skew. However, as we will show in this paper, there is room for improvement regarding the local skew. This is not really surprising, as FTSP and similar protocols work on a spanning tree, synchronizing nodes in the tree with their parents, and ultimately with the root of the tree. Neighboring nodes which are not closely related in the tree, i.e., where the closest common ancestor even is the root of the tree, will not be synchronized well because errors propagate down differently on different paths of the tree, see Figure 1. Eliminating all the deterministic sources of errors, the remaining two-hop error would be totally symmetric in the best case [23]. Indeed, every hop will experience some kind of inevitable random error $\delta$. As randomly distributed errors sum up according to the square-root function on each hop, the expected error between head and tail of a chain of $k$ nodes is in the order of $\delta\sqrt{k}$. Therefore, two nodes that are not in the same subtree rooted at the reference node are expected to experience an error in the order of the square-root of their distance in the tree.

In the theory community, clock synchronization has been studied for many years, recently with a focus on the local (also known as *gradient*) clock skew, e.g., [6, 12]. The goal of this paper is to investigate whether these theoretical insights carry over to practice. In particular, in Sections 4 and 5, we will propose the Gradient Time Synchronization Protocol (GTSP), a clock synchronization protocol that excels primarily at local clock synchronization. It is inspired by a long list of theoretical papers, originating in the distributed computing community [9, 13, 24, 17], lately also being adopted by the control theory community [21]. As such, GTSP is completely distributed, relying only on local information, requiring no reference node or tree construction. We argue that this approach results in a better average synchronization between neighbors while still maintaining a tolerable global skew. A thorough evaluation of our algorithm is performed on a testbed of Mica2 sensor nodes, and by simulations (Sections 6-9).

## 2. RELATED WORK

Clearly, clock synchronization has been studied extensively, long before the advent of wireless sensor networks. The classic solution is an atomic clock, such as in the global positioning system (GPS). Equipping each sensor node with a GPS receiver is feasible, but there are limitations in the form of cost and energy. Moreover, line of sight to the GPS satellites is needed, limiting the use to outdoor applications.

Classical clock synchronization algorithms rely on the ability to exchange messages at a high rate which may not be possible in wireless sensor networks. Traditional time synchronization algorithms like the *Network Time Protocol (NTP)* [16] are due to their complexity not well suited for sensor network applications. Moreover, as their application domain is different, they are not accurate enough for our purpose, even in a LAN they may experience skew in the order of milliseconds.

Sensor networks require sophisticated algorithms for clock synchronization since the hardware clocks in sensor nodes are often simple and may experience significant drift. Also, in contrast to wired networks, the multi-hop character of wireless sensor networks complicates the problem, as one cannot simply employ a standard client/server clock synchronization algorithm.

As research in sensor networks evolved during the last years, many different approaches for time synchronization were proposed. Römer presents a system [19] where events are time-stamped with the local clock. When such a timestamp is passed to another node, it is converted to the local timestamp of the receiving node.

*Reference Broadcast Synchronization (RBS)* [5] exploits the broadcast nature of the physical channel to synchronize a set of receivers with one another. A reference node is elected within each cluster to synchronize all other nodes. Since differences in the propagation times can generally be neglected in sensor networks, a reference message arrives at the same instant at all receivers. The timestamp of the reception of a broadcast message is recorded at each node and exchanged with other nodes to calculate relative clock offsets. RBS is designed for single-hop time synchronization only. However, nodes which participate in more than one cluster can be employed to convert the timestamps between local clock values of different clusters. Pulses from an external clock source attached to one node, for example a GPS receiver, can be treated like reference broadcasts to transform the local timestamps into UTC.

The *Timing-sync Protocol for Sensor Networks (TPSN)* [7] aims to provide network-wide time synchronization. The TPSN algorithm elects a root node and builds a spanning tree of the network during the initial level discovery phase. In the synchronization phase of the algorithm, nodes synchronize to their parent in the tree by a two-way message exchange. Using the timestamps embedded in the synchronization messages, the child node is able to calculate the transmission delay and the relative clock offset. However, TPSN does not compensate for clock drift which makes frequent resynchronization mandatory. In addition, TPSN causes a high communication overhead since a two-way message exchange is required for each child node.

These shortcomings are tackled by the *Flooding-Time Synchronization Protocol (FTSP)* [14]. A root node is elected which periodically floods its current timestamp into the network forming an ad-hoc tree structure. MAC layer time-stamping reduces possible sources of uncertainty in the message delay. Each node uses a linear regression table to convert between the local hardware clock and the clock of the reference node. The root node is dynamically elected by the network based on the smallest node identifier. After initialization, a node waits for a few rounds and listens for synchronization beacons from other nodes. Each node sufficiently synchronized to the root node starts broadcasting its estimation of the global clock. If a node does not receive synchronization messages during a certain period, it will declare itself the new root node.

The *Routing Integrated Time Synchronization protocol (RITS)* [20] provides post-facto synchronization. Detected events are time-stamped with the local time and reported to the sink. When such an event timestamp is forwarded towards the sink node, it is converted from the local time of the sender to the receiver's local time at each hop. A skew compensation strategy improves the accuracy of this approach in larger networks.

A completely distributed synchronization algorithm was proposed in [25]. The *Reachback Firefly Algorithm (RFA)* is inspired from the way neurons and fireflies spontaneously synchronize. Each node periodically generates a pulse (message) and observes pulses from other nodes to adjust its own firing phase. The authors report that a synchronization accuracy of 100µs can be achieved with this approach. RFA only provides synchronicity, nodes agree on the firing phases but do not have a common notion of time. Another shortcoming of RFA is the fact that it has a high communication overhead.

The fundamental problem of clock synchronization has been studied extensively and many theoretical results have been published which give bounds for the clock skew and communication costs [13, 17]. Srikanth and Toueg [24] presented a clock synchronization algorithm which minimizes the global skew, given the hardware clock drift.

The *gradient clock synchronization problem* was first introduced by Fan and Lynch in [6]. The gradient property of a clock synchronization algorithm requires that the clock skew between any two nodes is bounded by the distance (uncertainty in the message delay) between the two nodes. They prove a lower bound for the clock skew of $\Omega(d + \frac{\log D}{\log \log D})$ for two nodes with distance $d$, where $D$ is the network diameter. This lower bound also holds if delay uncertainties are neglected and an adversary can decide when a sync message will be sent [15]. Recently, Lenzen et al. [10] proposed a distributed clock synchronization algorithm guaranteeing clock skew $\mathcal{O}(\log D)$ between neighboring nodes while the global skew between any two nodes is bounded by $\mathcal{O}(D)$.

## 3. SYSTEM MODEL

In this section, we introduce the system model used throughout the rest of this paper. We assume a network consisting of a number of nodes equipped with a hardware clock subject to clock drift. Furthermore, nodes can convert the current hardware clock reading into a logical clock value and vice versa.

### 3.1 Hardware Clock

Each sensor node $i$ is equipped with a hardware clock $H_i(\cdot)$. The clock value at time $t$ is defined as

$$H_i(t) = \int_{t_0}^{t} h_i(\tau)\, d\tau + \Phi_i(t_0)$$

where $h_i(\tau)$ is the hardware clock rate at time $\tau$ and $\Phi_i(t_0)$ is the hardware clock offset at time $t_0$.

It is assumed that hardware clocks have bounded drift, i.e., there exists a constant $0 \leq \rho < 1$ such that

$$1 - \rho \leq h(t) \leq 1 + \rho$$

for all times $t$. This implies that the hardware clock never stops and always makes progress with at least a rate of $1 - \rho$. This is a reasonable assumption since common sensor nodes are equipped with external crystal oscillators which are used as clock source for a counter register of the microcontroller. These oscillators exhibit drift which is only gradually changing depending on the environmental conditions such as ambient temperature or battery voltage and on oscillator aging. This allows to assume the oscillator drift to be relatively constant over short time periods. Crystal oscillators used in sensor nodes normally exhibit a drift between 30 and 100 ppm.[1]

### 3.2 Logical Clock

Since other hardware components may depend on a continuously running hardware clock, its value should not be adjusted manually. Instead, a logical clock value $L_i(\cdot)$ is computed as a function of the current hardware clock. The logical clock value $L_i(t)$ represents the synchronized time of node $i$. It is calculated as follows:

$$L_i(t) = \int_{t_0}^{t} h_i(\tau) \cdot l_i(\tau)\, d\tau + \theta_i(t_0)$$

where $l_i(\tau)$ is the *relative logical clock rate* and $\theta_i(t_0)$ is the clock offset between the hardware clock and the logical clock at the reference time $t_0$. The logical clock is maintained as a software function and is only calculated on request based on a given hardware clock reading.

[1]ppm = parts per million. An oscillator with 100 ppm running at 1 MHz drifts apart 100µs in one second.

## 4. SYNCHRONIZATION ALGORITHM

In this section, we describe our distributed clock synchronization algorithm. The basic idea of the algorithm is to provide precise clock synchronization between direct neighbors while each node can be more loosely synchronized with nodes more hops away.

In a network consisting of sensor nodes with perfectly calibrated clocks (no drift), time progresses at the same rate throughout the network. It remains to calculate once the relative offsets amongst the nodes, so that they agree on a common global time. However, real hardware clocks exhibit relative drift in the order of up to 100 ppm leading to a continually increasing synchronization error between nodes.

Therefore, it is mandatory to repeat the synchronization process frequently to guarantee certain bounds for the synchronization error. However, precisely synchronized clocks between two synchronization points can only be achieved if the relative clock drift between nodes is compensated. In structured clock synchronization algorithms all nodes adapt the rate of their logical clock to the hardware clock rate of the reference node. This approach requires that a root node is elected and a tree structure of the network is established. Synchronization algorithms operating on structured networks have to cope with topology changes due to link failures or node mobility.

In a clock synchronization algorithm which should be completely distributed and reliable to link and node failures, it is not practicable to synchronize to the clock of a reference node. Therefore, our clock synchronization algorithm strives to agree with its neighbors on the current logical time. Having synchronized clocks is a twofold approach, one has to agree both on a common logical clock rate and on the absolute value of the logical clock.

### 4.1 Drift Compensation

We define the *absolute logical clock rate* $x_i(t)$ of node $i$ at time $t$ as follows:

$$x_i(t) = h_i(t) \cdot l_i(t)$$

Each node $i$ periodically broadcasts a synchronization beacon containing its current logical time $L_i(t)$ and the relative logical clock rate $l_i(t)$. Having received beacons from all neighboring nodes during a synchronization period, node $i$ uses this information to update its absolute logical clock rate as follows:

$$x_i(t_{k+1}) = \frac{\left(\sum_{j \in \mathcal{N}_i} x_j(t_k)\right) + x_i(t_k)}{|\mathcal{N}_i| + 1} \qquad (1)$$

where $\mathcal{N}_i$ is the set of neighbors of node $i$.

It is important to note that in practice node $i$ is unable to adjust $x_i$ itself since it has no possibility to measure its own hardware clock rate $h_i$. Instead, it can only update its relative logical clock rate $l_i = \frac{x_i}{h_i}$ as follows:

$$l_i(t_{k+1}) = \frac{\left(\sum_{j \in \mathcal{N}_i} \frac{x_j(t_k)}{h_i(t_k)}\right) + l_i(t_k)}{|\mathcal{N}_i| + 1} \tag{2}$$

We have to show that using this update mechanism all nodes converge to a common logical clock rate $x_{ss}$ which means that:

$$\lim_{t \to \infty} x_i(t) = \lim_{t \to \infty} h_i(t) \cdot l_i(t) = x_{ss}, \forall i$$

We assume that the network is represented as a graph $G(V, E)$ with the nodes as vertices and edges between nodes indicating a communication link between the two nodes. Using matrix multiplication the update of the logical clock rates performed in Equation (1) can be written as:

$$x(t+1) = A(t) \cdot x(t)$$

where the vector $x = (x_1, x_2, \ldots, x_n)^T$ contains the logical clock rates of the nodes. The entries of the $n \times n$ matrix $A$ are defined in the following way:

$$a_{ij} = \begin{cases} \frac{1}{|\mathcal{N}_i|+1} & \{i, j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

where $|\mathcal{N}_i|$ is the degree of node $i$. Since all rows of matrix $A$ sum up to exactly 1, it is *row stochastic.* Initially, the logical clock of each node $i$ has the same rate as the hardware clock ($x_i(0) = h_i(0)$) since the logical clock is initialized with $l_i(0) = 1$. It can be shown that all the logical clock rates will converge to a steady-state value $x_{ss}$:

$$\lim_{t \to \infty} x(t) = x_{ss}\mathbf{1} \tag{3}$$

The convergence of Equation (3) depends on whether the product $\prod_{t=0}^{\infty} A(t)$ of non-negative stochastic matrices has a limit. It is well-known that the product of row stochastic matrices converges if the graph corresponding to matrices $A(t)$ is strongly connected [26, 4].

## 4.2 Offset Compensation

Besides having all nodes agreed on the rate the logical clock is advanced, it is also necessary to synchronize the actual clock values itself. Again, the nodes have to agree on a common clock value, which can be obtained by calculating the average of the clock values as for the drift compensation. A node $i$ updates its logical clock offset $\theta_i$ as follows:

$$\theta_i(t_{k+1}) = \theta_i(t_k) + \frac{\sum_{j \in \mathcal{N}_i} L_j(t_k) - L_i(t_k)}{|\mathcal{N}_i| + 1} \tag{4}$$

However, using the average of all neighbors as the new clock value is problematic if the offsets are large. During node startup, the hardware clock register is initialized to zero, resulting possibly in a huge offset to nodes which are already synchronized with the network. Such a huge offset would force all other nodes to turn back their clocks which violates the causality principle. Instead, if a node learns that a neighbor's clock is further ahead than a certain threshold value, it jumps to the neighbors clock value.

By employing this bootstrap mechanism, a node joining the network gets synchronized quickly with the rest of the network. In the worst case it can take up to $\mathcal{O}(D)$ time to have all nodes loosely synchronized, where $D$ is the diameter of the network. Since the logical clock rate of a node which recently joined the network may not be synchronized with the network yet, its clock value will start to drift apart immediately after the initial synchronization point. The resulting synchronization error is bounded by the hardware clock drift accumulated during a synchronization interval.

## 4.3 Computation and Memory Requirements

Computation of the logical clock rate involves floating point operations. Since most sensor platforms support integers only, floating point arithmetic has to be emulated using software libraries which are computation intensive. However, since the range of the logical clock rate is bounded by the maximum clock drift, computations can greatly benefit from the use of fixed point arithmetic.

Besides the computational constrains of current sensor hardware, data memory is also very limited and the initial capacity of data structures has to be specified in advance. The synchronization algorithm requires to store information about the relative clock rates of its neighbors which are used in Equation (2). Since the capacity of the data structures is limited, the maximal number of neighbors a node accounts for in the calculations is also limited and a node possibly has to discard crucial neighbor information. However, ignoring messages from a specific neighbor does still lead to consensus as long as the resulting graph remains strongly connected. Since the capacity constraints are only a problem in very dense networks, it is very unlikely that a partitioning of the network graph is introduced.

## 4.4 Energy Efficiency

Radio communication consumes a large fraction of the energy budget of a sensor node. While the microcontroller can be put into sleep mode when it is idle, thus reducing the power consumption by a large factor, the radio module still needs to be powered to capture incoming message transmissions. Energy-efficient communication protocols, e.g., [18], employ scheduled radio duty-cycling mechanisms to lower the power consumption and thus prolonging battery lifetime. Since the exact timing when synchronization messages are sent is not important, GTSP can be used together with an energy-efficient communication layer. In addition, a node can estimate the current synchronization error to its neighbors from the incoming beacons in order to dynamically adapt the interval between synchronization beacons. If the network is well synchronized, the beacon rate can be lowered to save energy. The communication overhead of GTSP is comparable with FTSP since both algorithms require each node to broadcast its time information only once during a synchronization period.

## 5. IMPLEMENTATION

This section describes the implementation of our gradient clock synchronization algorithm on the Mica2 sensor nodes using the TinyOS operating system.

### 5.1 Target Platform

The hardware platform used for the implementation of the algorithm is the Mica2 sensor node from Crossbow. It features an ATmega128L low-power microcontroller from Atmel with 4 kB of RAM, 128 kB program ROM and 512 kB external flash storage. The CC1000 radio module has been designed for low-power applications and offers data rates up to 76.8 kBaud using frequency shift keying (FSK).

The ATmega128L microcontroller has two built-in 8-bit timers and two built-in 16-bit timers. The Mica2 board is equipped with two different quartz oscillators (32 kHz and 7.37 MHz) which can be used as clock sources for the timers. Timer3 is configured to operate at 1/8 of the oscillator frequency (7.37 MHz) leading to a clock frequency of 921 kHz. Since Timer3 is sourced by an external oscillator it is also operational when the microcontroller is in low-power mode. We employ Timer3 to provide our system with a free-running 32-bit hardware clock which offers a precision of a microsecond. This approach on the Mica2 node offers better clock granularity as compared to more recent hardware platforms which lack a high frequency external oscillator, see Table 1.

| Platform | CPU clock | Quartz crystal |
|---|---|---|
| Mica2 | 8 MHz | 32 kHz, 7.37 MHz |
| IRIS | 8 MHz | 32 kHz, 7.37 MHz |
| TinyNode | 8 MHz | 32 kHz |
| Tmote Sky | 8 MHz | 32 kHz |

**Table 1: Comparison of clock sources for common sensor network hardware platforms.**

### 5.2 TinyOS Implementation

The implementation of GTSP on the Mica2 platform is done in TinyOS 2.1. The protocol implementation provides time synchronization as service for an application running on the mote. The architecture of the time synchronization component and its relation to other system components is shown in Figure 2.

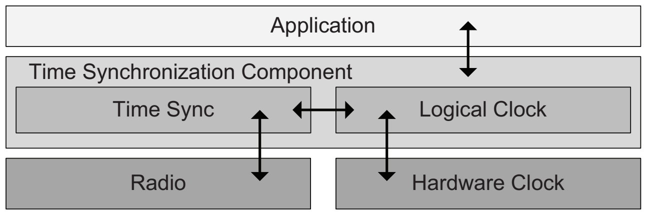


**Figure 2: Architecture of the time synchronization service and its integration within the hardware and software platform. Arrows indicate the flow of information between different components.**

The *TimeSync* module periodically broadcasts a synchronization beacon containing the current logical time $L_i(t)$ and the relative logical clock rate $l_i(t)$. Each node is overhearing messages sent by neighboring nodes. The timestamp contained in the synchronization beacons is used to update the current offset between the hardware and the logical time and the rate of the logical clock according to Equations (2) and (4). The hardware and logical time when the most recent synchronization beacon of each neighbor has been received is stored in a neighbor table.

By overhearing synchronization beacons a node will learn when a node joins its neighborhood. When no beacon messages were received from a node for several consecutive beacon intervals, the link to this node is assumed to be broken and the node is removed from the neighbor table. The capacity of the neighbor table is limited by the data memory available on the node. An upper bound for the required capacity is the maximum node degree in the network. However, as long as the resulting network graph stays connected it is possible to ignore synchronization beacons from a specific neighbor. The default capacity of the neighbor table in our implementation is set to 16.

Furthermore, the time interval between synchronization beacons can be adapted dynamically. This allows to increase the frequency of beacons during the bootstrap phase or when a new node has recently joined the network. On the other side, if the system is in the steady-state, i.e., all nodes are quite well synchronized to their neighbors, reducing the number of sent beacons can save energy.

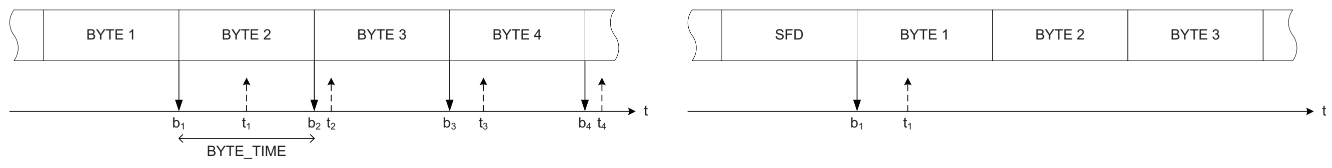


**Figure 3: Timestamping at the MAC Layer: An interrupt (solid arrow) is generated if a complete byte is received by the CC1000 radio chip. Dashed arrows indicate the time when the interrupt handler takes the timestamp for the current byte (left). Packet oriented radio chips like the CC2420 generate a single interrupt when the start frame delimiter (SFD) has been received (right).**

## 5.3 MAC Layer Timestamping

Broadcasting time information using periodic beacons is optimal in terms of the message complexity since the neighbor is not forced to acknowledge the message as in sender-receiver synchronization schemes (e.g., TPSN). However, the propagation delay of a message cannot be calculated directly from the embedded timestamps. Exchanging the current timestamp of a node by a broadcast introduces errors with magnitudes larger than the required precision due to non-determinism in the message delay. The time it takes from the point of time where the message is passed to the communication stack until it reaches the application layer on a neighboring node is highly non-deterministic due to various sources of errors induced in the message path [8, 7]. Reducing the main sources of errors by time-stamping at the MAC layer is a well-known approach, e.g., the FTSP time-stamping scheme [14]. The current timestamp is written into the message payload right before the packet is transmitted over the air. Accordingly, at the receiver side the timestamp is recorded right after the preamble bytes of an incoming message have been received.

Byte-oriented radio chips, e.g., the CC1000 chip of the Mica2 platform, generate an interrupt when a complete data byte has been received and written into the input buffer. The interrupt handler reads the current timestamp from the hardware clock and stores it in the metadata of the message. However, there exists some jitter in the reaction time of the interrupt handler for incoming radio data bytes.

The concurrency model of TinyOS requires that asynchronous access to shared variables has to be protected by the use of `atomic` sections [11]. An interrupt signaled during this period is delayed until the end of the atomic block. To achieve clock synchronization with accuracy in the order of a few microseconds, it is inevitable to cope with such cases in order to reduce the variance in the message delay. Therefore, each message is time-stamped multiple times both at the sender and receiver sides.

The radio chip generates an interrupt at time $b_i$ when a new data byte has arrived or is ready to be transmitted. The interrupt handler is invoked and reads the current hardware clock value at time $t_i$ as shown in Figure 3. The time it takes the radio chip to transmit a single byte over the air is denoted by the BYTE_TIME. This constant can be calculated directly from the baud rate and encoding settings of the radio chip. Due to the fact that it takes BYTE_TIME to transmit a single byte, the following equation holds for all timestamps:

$$b_{i-1} \leq t_i - \text{BYTE\_TIME}$$

Using multiple timestamps, it is hence possible to compensate for the interrupt latency. A better estimation for the timestamp of the $i$-th byte can calculated as follows:

$$t'_i = \min(t_i, t'_{i+1} - \text{BYTE\_TIME})$$

The timestamps of the first six bytes are used to estimate the arrival time of a packet. A single timestamp for this packet is then calculated by taking the average of these timestamps. Packet-oriented radio chips as the CC2420 (MicaZ or TmoteSky) or the RF230 (IRIS mote) unburden the microcontroller from handling every byte transmission separately. Instead, a single interrupt is generated when the start frame delimiter (SFD) has been received. Subsequent bytes of the payload are written directly into the FIFO receive buffer. Therefore, compensating jitter in the interrupt handling time is not possible with packet-oriented radio chips.

Three Mica2 nodes were used to calibrate the MAC layer time-stamping. One node is continuously transmitting messages to the receiver node. Both nodes raise an output pin when the interrupt handler responsible for the time-stamping is executed. This corresponds to the points in time when a byte is time-stamped. The output pins are connected by wires to the input pins of a third node which is configured to trigger an interrupt on a rising edge. The time difference between the send and receive interrupts corresponds to the transmission delay. In this measurement setup, the propagation delay is ignored since it is very small for typical sensor networks, i.e., less than 1µs for a distance of 300 meters. By exchanging roughly 70,000 calibration packets, an average transmission delay of 1276 clock ticks with a standard deviation of 1.95 ticks was observed. Figure 4 shows the variance observed in the measurements of the transmission delay. It can be clearly seen that large errors in the transmission delay are introduced without a sophisticated mechanism to compensate for the latency in the interrupt handling.

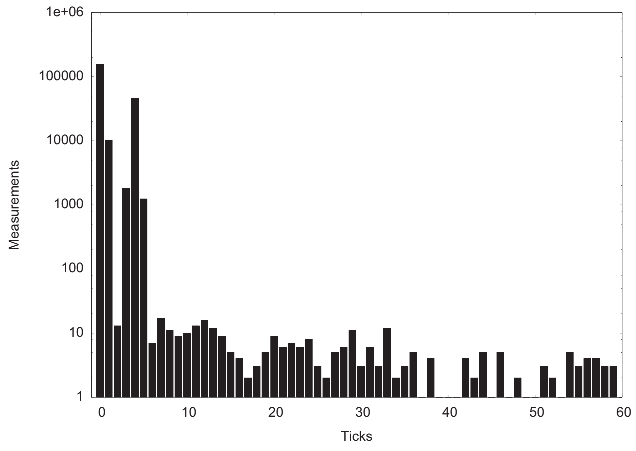


**Figure 4: Measurements of the latency in the interrupt handling for the Mica2 node.**

## 6. EVALUATION

In the following sections of this paper, we evaluate the performance of the Gradient Time Synchronization Protocol (GTSP). Evaluating clock synchronization algorithms is always an issue since various performance aspects can be evaluated, e.g., precision, energy consumption, or communication overhead. In this paper, we restrict our evaluation to the precision achieved by the synchronization algorithm. Measuring the instantaneous error between logical clock of different nodes is only possible at a common time instant, e.g., when all nodes can observe the same event simultaneously. A general practice when evaluating time synchronization algorithms for sensor networks is to transmit a message as a reference broadcast. All nodes are placed in communication range of the reference broadcaster. The broadcast message arrives simultaneously at all nodes (if the minimal differences in the propagation delay are neglected) and is time-stamped with the hardware clock. The corresponding logical clock value is used to calculate the synchronization error to other nodes. Two different metrics are used throughout the evaluation in this paper: the *Average Neighbor Error* measures the average pair-wise differences in the logical clock values of nodes which are *direct* neighbors in the network graph while the *Average Network Error* is defined as the average synchronization error between *arbitrary* nodes.

## 7. TESTBED EXPERIMENTS

We evaluated the implementation of GTSP by experiments on a testbed which consists of 20 Mica2 sensor nodes. Experiments with the identical setup are also performed for FTSP which is the standard time synchronization protocol in TinyOS. All nodes are placed in close proximity forming a single broadcast domain. In addition, a base station node is attached to a PC to log synchronization messages sent by the nodes. To facilitate measurements on different network topologies, a virtual network layer is introduced in the management software of the sensor nodes. Each node can be configured with a whitelist of nodes from which it will further process incoming messages, packets from all other nodes are ignored. Using this virtual network layer different network topologies can be enforced by software.

The base station periodically broadcasts probe messages to query the current logical time of all the nodes. The interval between time probes is uniformly distributed between 18 and 22 seconds. To reduce radio collisions with time synchronization messages, nodes do not reply with the current time value. Instead, the current local timestamp and the estimated logical timestamp are logged to the external flash memory.

### 7.1 Experimental Results for GTSP

At the begin of the experiment, the configuration parameters for GTSP were set for all nodes. The synchronization algorithm was started on every node at random during the first 30 seconds of the experiment. Synchronization beacons are broadcasted every 30 seconds. The offset threshold parameter is set to 10. Therefore, a node adjust its logical clock value if the logical clock of a neighbor is further ahead than 10µs. Right after the initialization all nodes have zero logical clock offset and the rate of the logical clock corresponds to the hardware clock rate. We denote the period between synchronization beacons by $P$ and the network diameter by $D$. It takes up to $D \cdot P$ time until all nodes raised their logical clock to the value of the node having the highest hardware clock value. After having received the second beacon from a neighboring node, nodes can estimate the rate of the neighbor's logical clock (relative to the local hardware clock). To reduce the effects of jitter in the message delay, the estimated clock rates of the neighbors are filtered by a moving average filter with $\alpha = 0.6$. The experiments lasted for approximately 6 hours which resulted in around 1000 time probes logged to the flash storage of the sensor nodes. The measurement results for GTSP on a ring of 20 Mica2 nodes is depicted in Figure 5. It can be seen that GTSP achieves an average synchronization error between neighbors of 4.0µs after the initialization phase has been completed ($t > 5000s$). The average network synchronization error is 14.0µs for

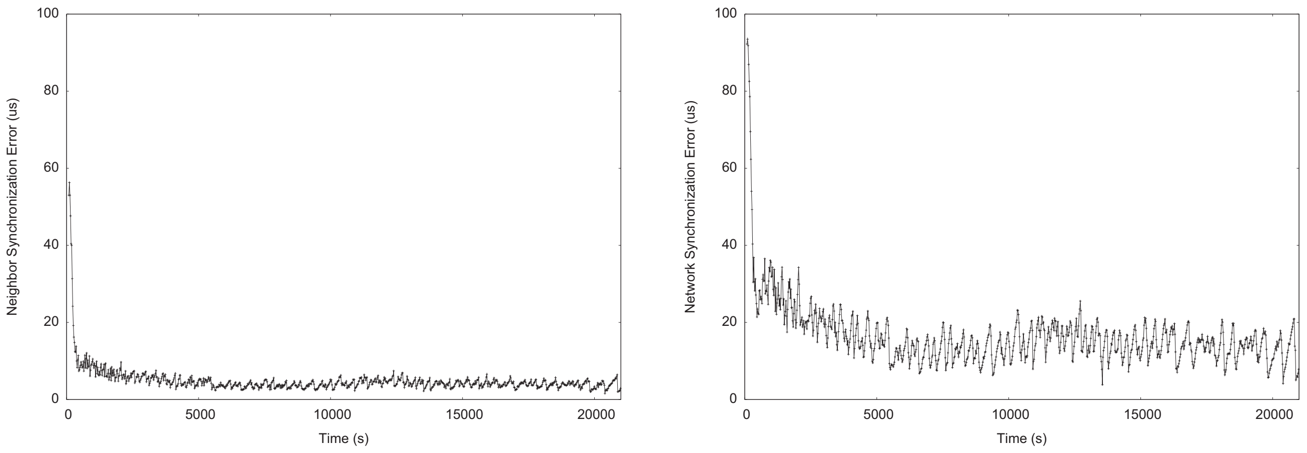


**Figure 5: Average neighbor (4.0µs) and network synchronization errors (14.0µs) measured for GTSP on a ring of 20 Mica2 nodes.**

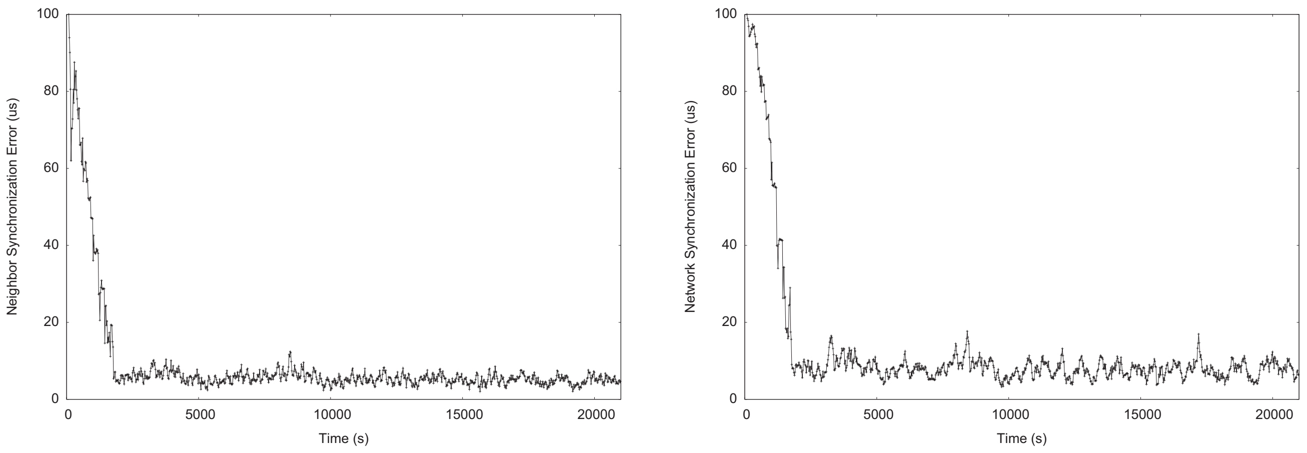
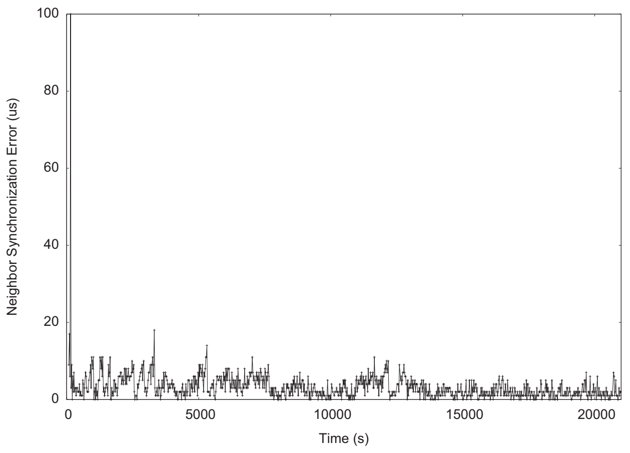


**Figure 6: Average neighbor (5.3µs) and network synchronization errors (7.7µs) measured for FTSP on a ring of 20 Mica2 nodes.**

the same interval. For our testbed setup consisting of 20 nodes placed in a ring, it takes roughly 30 minutes until the algorithm converges which is comparable to the convergence time of FTSP, see next subsection.

## 7.2 Comparison with FTSP

The same network topology was used to compare the performance of GTSP with FTSP which is considered to be the state-of-the-art time synchronization protocol for wireless sensor networks. The default parameter settings from TinyOS 2.1 were used for FTSP, see Table 2. The measurement results for FTSP on a ring of 20 nodes are shown in Figure 6. The time it takes FTSP to synchronize all nodes to the reference node highly depends on the network diameter and the placement of nodes in the network. Again, the time synchronization algorithm is started on all nodes in a random sequence during the first 30 seconds of the experiment. Newly initialized nodes do not send synchronization beacons during an initial period which is determined by the `ROOT_TIMEOUT` parameter. If no other beacons are received during that period, a node declares itself as the new root node and starts broadcasting beacons. Therefore, multiple root nodes are present right after the beginning of the experiment. When a node learns about another root node with a lower identifier than the current root, it switches its root node and adapts its regression table to the logical time of the new root node. If the regression table contains more than `ENTRY_SEND_LIMIT` entries, the node retransmits the logical clock of its current root node. Due to this behavior of FTSP, it takes roughly 30 minutes until all nodes are synchronized to a common logical clock in our setup. We argue that GTSP provides better synchronized clocks during the initialization phase of the algorithm compared to FTSP since clock values are propagated immediately through the



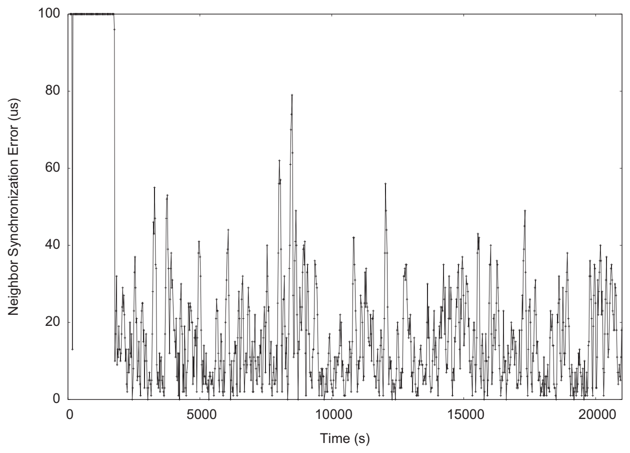


**Figure 7: Neighbor synchronization error between Node 8 and Node 15 on the ring of Figure 8 for GTSP (*left*) and FTSP (*right*). GTSP achieves an average error of 2.8µs with a standard deviation of 2.1µs for t>5000s. FTSP achieves an average error of 15.0µs with a standard deviation of 12.4µs for t>5000s.**

network. Although not in the focus of this paper, this may be an advantage of GTSP in dynamic networks. After FTSP has converged at $t > 5000s$, we measured an average neighbor synchronization error of 5.3µs and a network error of 7.7µs.

| Protocol parameter | Value |
|---|---|
| Synchronization period | 30s |
| ROOT_TIMEOUT | 5 |
| IGNORE_ROOT_MSG | 4 |
| ENTRY_SEND_LIMIT | 3 |

**Table 2: Protocol parameters for FTSP.**

FTSP implicitly creates an ad-hoc tree on the network graph by flooding the network with the logical time of the root. Only synchronization beacons containing a higher sequence number are added to the regression table, other packets are ignored. Therefore, the ring network depicted in Figure 8 is split into two subtrees rooted at Node 1. The leaves of these subtrees are Node 8 and Node 15, respectively. Although Node 8 is receiving synchronization beacons from Node 15, this time information is ignored since it contains the same sequence number as previously received from Node 20. Therefore, Node 8 and Node 15 do not synchronize to each other in contrast to the local synchronization approach presented in GTSP. Figure 7 shows the synchronization error between Node 8 and Node 15 for both protocols. Our measurement results show that GTSP provides a better neighbor synchronization compared to FTSP. One might argue that this ring example looks "cooked-up" and that a ring topology does not happen often in practice. While this is true, we insist that the point we make is valid in general, as many reasonable network topologies (e.g., uniform random distribution, grid topology) do not allow a tree embedding with low stretch. In any sensible network topology, FTSP will have neighboring nodes that have a tree distance in the order of the diameter of the network. Therefore, the effects shown in Figure 7 will always occur in real-world network topologies even though at a smaller scale. Experiments on a 4x5 grid topology with FTSP showed that neighboring nodes can have a large stretch (e.g., we experienced a stretch up to 13) when the node identifiers are assigned randomly and nodes were started in a random order.

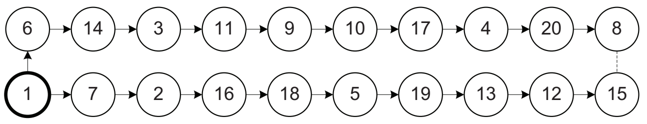


**Figure 8: The ring synchronization problem: Although Node 8 and Node 15 are direct neighbors in the ring, they are leaves of two different subtrees rooted at Node 1.**

## 8. SIMULATIONS

In addition to the experiments performed on the testbed, we implemented GTSP in a network simulator [1]. Simulating an algorithm can never supersede an experimental evaluation on a testbed since it is infeasible to simulate the exact behavior of the hardware (e.g., interrupt latency, interferences). However, simulations are a good way to gain a first impression on how the algorithm performs on a large scale network.

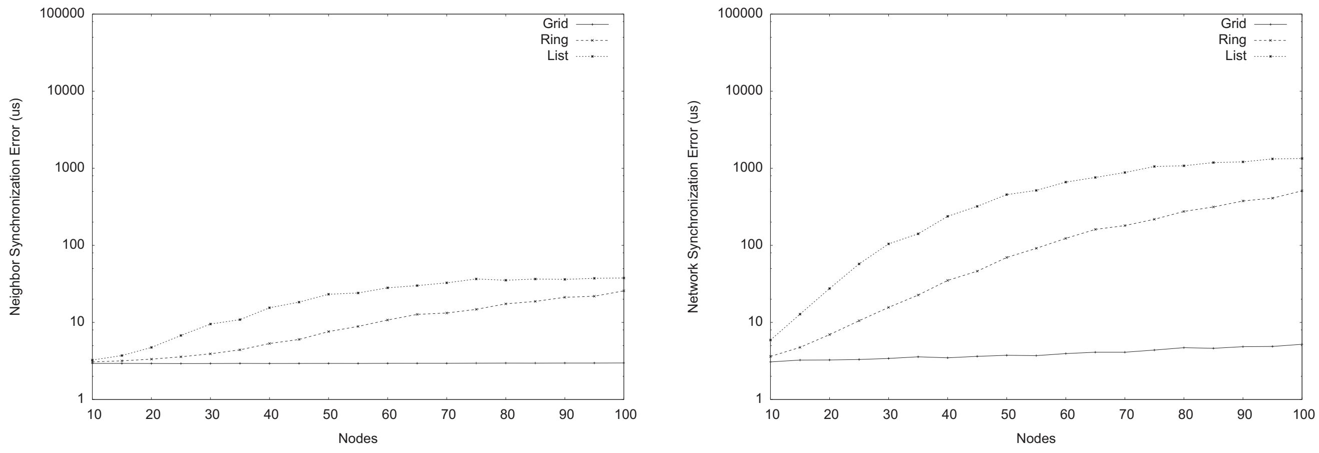


**Figure 9: Average neighbor (left) and network synchronization errors (right) measured by simulations of the Gradient Time Synchronization Protocol (GTSP) on different network topologies.**

For the simulation of the sensor nodes, we modeled the hardware clock of a node in software. At the start of a simulation run, each node is initialized with a random hardware clock drift of 30 ppm and a random start value. Although MAC layer time-stamping schemes (see Section 5.3) can reduce large variances in the transmission delay, there always remains some jitter in the message delay which affects the time synchronization. For the simulations the variances in the message delay are modeled by a normally distributed random variable with zero mean and a standard deviation of 2. We measured the average synchronization error between neighbors and the average network-wide synchronization error for different network topologies. For each network setting we averaged the errors over 10 different simulation runs. The results are depicted in Figure 9.

The algorithm performs best when the nodes form a grid which has the smallest network diameter amongst the studied topologies. Not surprisingly, the worst clock accuracy is achieved when the nodes are placed in a line, forming a network with maximal diameter. The simulation results clearly show that the synchronization error between neighbors is increasing with the network diameter.

## 9. CONCLUSION AND FUTURE WORK

Sensor network applications can greatly benefit from synchronized clocks to perform data fusion or energy-efficient communication. A perfect clock synchronization algorithm should fulfill a handful of different properties at the same time: precise global and local time synchronization, fast convergence, fault-tolerance, and energy-efficiency. Classical time synchronization algorithms used in wireless sensor networks strive to optimize the global clock skew. However, we argue that many practical applications will benefit from minimizing local clock skew.

In this paper, we presented the Gradient Time Synchronization Protocol (GTSP) which is a completely distributed time synchronization protocol. Nodes periodically broadcast synchronization beacons to their neighbors. Using a simple update algorithm, they try to agree on a common logical clock with their neighbors. It can be shown by theoretical analysis that by employing this algorithm, the logical clock of nodes converge to a common logical clock. GTSP relies on local information only, making it robust to node failures and changes in the network topology.

Experiments on a testbed setup of 20 Mica2 nodes and simulations showed that the remaining synchronization error between neighbors is small while still maintaining an acceptable global skew. Furthermore, we have shown that GTSP can improve the synchronization error between neighboring sensor nodes compared to tree-based time synchronization protocols.

The goal of this paper is to bridge the gap between theory and practice in the area of clock synchronization for sensor networks. The proposed time synchronization protocol is intended to be used as the ground for further research in this area.

## 10. ACKNOWLEDGMENTS

We would like to thank Nicolas Burri, Branislav Kusy and Christoph Lenzen for their comments and suggestions which helped us to improve this paper. Furthermore, we thank the anonymous reviewers for their valuable comments.

## 11. REFERENCES

[1] Sinalgo - Simulator for Network Algorithms. http://dcg.ethz.ch/projects/sinalgo/.

[2] M. Allen, L. Girod, R. Newton, S. Madden, D. T. Blumstein, and D. Estrin. VoxNet: An Interactive, Rapidly-Deployable Acoustic Monitoring Platform. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, 2008.

[3] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: Ultra-low Power Data Gathering in Sensor Networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, 2007.

[4] M. Cao, A. S. Morse, and B. D. O. Anderson. Reaching a Consensus in a Dynamically Changing Environment: Convergence Rates, Measurement Delays, and Asynchronous Events. *SIAM J. Control Optim.*, 47(2), 2008.

[5] J. Elson, L. Girod, and D. Estrin. Fine-Grained Network Time Synchronization using Reference Broadcasts. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[6] R. Fan and N. Lynch. Gradient Clock Synchronization. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, 2004.

[7] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync Protocol for Sensor Networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003.

[8] H. Kopetz and W. Ochsenreiter. Clock Synchronization in Distributed Real-Time Systems. *IEEE Trans. Comput.*, 36(8), 1987.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[10] C. Lenzen, T. Locher, and R. Wattenhofer. Clock Synchronization with Bounded Global and Local Skew. In *49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2008.

[11] P. Levis. TinyOS Programming. http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf.

[12] T. Locher and R. Wattenhofer. Oblivious Gradient Clock Synchronization. In *20th International Symposium on Distributed Computing (DISC)*, 2006.

[13] J. Lundelius and N. A. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984.

[14] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi. The Flooding Time Synchronization Protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[15] L. Meier and L. Thiele. Brief announcement: Gradient clock synchronization in sensor networks. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, 2005.

[16] D. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[17] R. Ostrovsky and B. Patt-Shamir. Optimal and Efficient Clock Synchronization Under drifting Clocks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, 1999.

[18] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[19] K. Römer. Time Synchronization in Ad Hoc Networks. In *MobiHoc '01: Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, 2001.

[20] J. Sallai, B. Kusy, A. Ledeczi, and P. Dutta. On the scalability of routing integrated time synchronization. *3rd European Workshop on Wireless Sensor Networks (EWSN)*, 2006.

[21] L. Schenato and G. Gamba. A distributed consensus protocol for clock synchronization in wireless sensor network. *46th IEEE Conference on Decision and Control*, 2007.

[22] G. Simon, M. Maróti, Á. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004.

[23] P. Sommer and R. Wattenhofer. Symmetric Clock Synchronization in Sensor Networks. In *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2008.

[24] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *J. ACM*, 34(3), 1987.

[25] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, and R. Nagpal. Firefly-Inspired Sensor Network Synchronicity with Realistic Radio Effects. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, 2005.

[26] J. Wolfowitz. Products of Indecomposable, Aperiodic, Stochastic Matrices. *Proceedings of the American Mathematical Society*, 14(5), 1963.