Elapsed Time on Arrival: A simple and versatile primitive for canonical time synchronization services

Branislav Kusý[†], Prabal Dutta[‡], Philip Levis[‡], Miklós Maróti[†], Ákos Lédeczi[†*], and David Culler[‡]

[†]Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee 37203, USA
E-mail: {branislav.kusy,miklos.maroti,akos.ledeczi}@vanderbilt.edu

[‡]Computer Science Division
University of California, Berkeley
Berkeley, California 94720, USA
E-mail: {prabal,pal,culler}@cs.berkeley.edu

[*]Corresponding author

Abstract:
Time synchronization is one of the most important and fundamental middleware services for wireless sensor networks. However, there is an apparent disconnect between existing time synchronization implementations and the actual needs of current typical sensor network applications. To address this problem, we formulate a set of canonical time synchronization services distilled from actual applications and propose a set of general application programming interfaces for providing them. We argue that these services can be implemented using a simple time-stamping primitive called *Elapsed Time on Arrival* (ETA) and we provide two such implementations. The Routing Integrated Time Synchronization (RITS) is an extension of ETA over multiple hops. It is a reactive time synchronization protocol that can be used to correlate multiple event detections at one or more locations to within microseconds. Rapid Time Synchronization (RATS) is a proactive timesync protocol that utilizes RITS to achieve network-wide synchronization with microsecond precision and rapid convergence. Our work demonstrates it is possible to build high-performance timesync services using the simple ETA primitive and suggests that more complex mechanisms may be unnecessary to meet the needs of many real world sensor network applications.

Keywords: Sensor Networks, Ad Hoc Networks, Ubiquitous Computing, Time Synchronization, Clock Synchronization, Clock Drift, Multi-hop, Medium Access Control, Packet Delay

1 Introduction

Time synchronization has been extensively studied in the context of distributed systems theory, internetworks, local area networks, real-time control systems, wireless sensor networks (WSNs), and many applications. These different contexts have led to different notions of time. A common notion of time allows us to reason about the ordering of events, the causal relationships and correlations between events, the rate of change of observations over time, and also enables the coordination of future actions. Events in distributed systems can be specified in the *temporal*, *causal*, *logical*, and *delivery* order, as well as expressed in *local* and *global* time frames (Kopetz, 1997; Lamport, 1978; Reichenbach, 1957).

In the context of WSNs, timesync commonly refers to the problem of synchronizing clocks across a set of sensor nodes which are connected to one another over a single- or multi-hop wireless communications channel. Many existing sensor network applications require temporal ordering of events either relative to other events or relative to an absolute external frame of reference. A variety of time synchronization algorithms have been proposed to address both the temporal ordering of events and the more general problem of establishing a virtual global timebase across a network. Often, these timesync algorithms require nodes to involve third parties (such as reference broadcasters), perform two-way handshakes, exchange multiple messages, or perform computationally expensive optimization, for example, to maintain synchronized clocks and a consistent sense of network time. Section surveys the related work on time synchronization in sensors networks, describes how they correlate with the canonical services and what sources of errors they eliminate.

Since all of these operations consume extra system resources, but not all applications have identical synchronization needs, we may find that system resources are being wasted unnecessarily. This suggests that these more complex and intricate mechanisms may be inappropriate to meet the needs of many common sensor network applications. We suggest that a better match between applications and timesync implementations is possible, but it requires a better understanding of the ways in which applications use timesync services. Section explores the time synchronization requirements of common sensor applications and distills a set of canonical timesync services from several application domains into a set of timesync application programming interfaces.

Section 3.4 analyzes the sources of time-stamping jitter and physical clock variation. We suggest several mechanisms that exploit the rich interfaces between the software and hardware layers to minimize the non-determinism. Our implementation of these mechanisms is a timestamping primitive called elapsed time on arrival (ETA), which is a medium access control layer plug-in that reduces or eliminates nearly all sources of time-stamping jitter.

Timesync services can utilize ETA as a low-level primitive provided by the operating system that abstracts underlying hardware and drivers. Porting a timesync service to a different hardware platform requires simply porting the ETA primitive to that platform, promoting loose-coupling and reuse of timesync code. To demonstrate the versatility of ETA, we implemented two general timesync services. Routing integrated time synchronization (RITS) is a reactive time synchronization protocol that can be used to correlate multiple event detections at one or more locations to within microseconds. Rapid Time Synchronization (RATS) is a proactive timesync protocol that achieves network-wide synchronization with microsecond precision and rapid convergence. The design details of ETA, RITS, and RATS is presented in Section 4.4 and a performance evaluation is presented in Section 5.3.

Conceptually similar timestamping primitives have been proposed in the literature but no earlier implementation has achieved comparable performance. As a consequence, other implementations have resorted to more complex designs to mitigate the effects of timing jitter. Our work suggests that these more complex timesync mechanisms may be unnecessary to meet the needs of many common sensor network applications.

2 Related Work

WSNs present a unique set of challenges not found in traditional networks, such as unreliable and possibly unidirectional links, dense and dynamically changing topology, and extremely limited energy, computational power and memory resources. Consequently, many results in the rich literature on time synchronization were not directly applicable in WSNs, and a number of new algorithms and protocols have been proposed.

Sivrikaya and Yener (2004) present a survey of sensor network time synchronization protocols. Their paper outlines the synchronization problems using computer clocks as the timebase, identifies common challenges for synchronization methods, and enumerates the requirements of synchronization schemes for sensor networks. One of these requirements, called *immediacy*, quantifies the allowable latency when reporting an event to a sink node. The paper claims that immediacy requirements might prevent the use of certain protocols which, the paper claims, in turn requires that nodes be *presynchronized* at all times. We show this claim to be inaccurate; that presynchronization is not necessary; and that our post-facto approach can convey the event time just as rapidly.

A more in-depth analysis of the timesync problem is presented by K. Römer (2005) which extends significantly the analysis of essential issues surrounding timesync for sensor networks presented by Elson and Romer (2002). This book chapter describes a system model, identifies classes of synchronization, presents synchronization techniques, reviews and classifies ten concrete timesync algorithms from the literature, proposes evaluation strategies, and casts time synchronization as a special case of the calibration

problem. Of particular interest to us are the descriptions, classifications, and evaluations of the ten timesync protocols: Time-stamp Synchronization (TSS) (Romer, 2004), Reference Broadcast Synchronization (RBS) (Elson, 2002), Tiny-Sync and Mini-Sync (TS/MS) (Sichitiu and Veerarittiphan, 2003), Lightweight Time Synchronization (LTS) (van Greunen and Rabaey, 2003), Timing-Sync Protocol for Sensor Networks (TPSN) (Ganeriwal et al., 2003), TSync (Dai and Han, 1994), Interval-Based Synchronization (IBS) (Marzullo and Owicki, 1983; Blum et al., 2004; Meier et al., 2004; Schmid and Schossmaier, 1997), Flooding Time Synchronization Protocol (FTSP) (Maróti et al., 2004), Asynchronous Diffusion (AD) (Li and Rus, 2004), and Time Diffusion Synchronization (TDP) (Su and Akyildiz, 2004). Most relevant to our work are TSS, RBS, TPSN, FTSP, and the timesync protocol used in a structural monitoring application by Xu et al. (2004).

The Time-Stamp Synchronization (TSS) approach is conceptually similar to our time-stamping primitive. One drawback to the TSS approach is the requirement for multiple floating-point operations at each node which, on mote-class devices, is expensive. Another, more significant drawback stems from the assumption that it is difficult to accurately estimate message delays between neighboring nodes. This assumption is due to the queuing, randomness, and contention that typically occurs in the media access control layer of many networks on desktop-class computers. However, with the advent of specially-designed operating systems such as TinyOS (Hill, 2000) and integrated sensor nodes like the Mica motes, event callbacks that are invoked at the *actual time of message transmission* are possible.

The Reference Broadcast System (RBS), perhaps the best known timesync algorithm in the sensor network regime, uses a transmitter node to synchronize the clocks of two receiver nodes to each other. The RBS approach time-stamps messages only on the receiver side, which eliminates the delays on the sender side, most notably in the medium access control layer. The accuracy of the RBS time-stamping reported by the authors is approximately 11μs on the Mica platform. Least square linear regression is used to account for the clock drifts which results in 7.4μs average error between two motes after a 60 second interval. The RBS solution for the multi-hop scenario uses concepts similar to one of our proposed canonical services. However, because of the extra coordination overhead of reference broadcasters, the implementation is considerably more complicated and less energy efficient than our proposed solution in Section 5.2.

The Timing-Sync Protocol for Sensor Networks (TPSN) eliminates sources of time-stamping errors by making use of the implicit acknowledgments to transmit information back to the sender. This protocol gains an additional accuracy over RBS due to time-stamping the radio message twice and averaging these time-stamps. Since TPSN relies on a two-way information exchange, messages cannot be broadcasted which results in higher communication load. The authors of TPSN algorithm implemented both TPSN and RBS on the Mica platform using a 4MHz clock for time-stamping, and compared the precision of the two algorithms. The resulting average errors for a single hop case for two nodes are 16.9μs and 29.1μs for the TPSN and RBS algorithms, respectively.

The Flooding Time Synchronization Protocol (FTSP) improves on the ideas in RBS and TPSN. It is a proactive protocol in which every node periodically broadcasts timesync messages. The local clock of a single dynamically elected node provides the global time in the network. A time synchronization hierarchy rooted at this node is created in which nodes are synchronized by receiving global time estimates from nodes closer to the root. The applied flood-based communication protocol in FTSP provides a very robust network, and still incurs only small network traffic. The algorithm uses a fine-grained clock, MAC-layer time-stamping with several jitter reducing techniques and clock drift estimation to achieve its high precision. On the Mica2 platform the reported average error of the algorithm is 1.48μs for the single hop case, and 0.5μs per hop in a 6-hop network for the multi-hop case. However, the initial convergence time of FTSP on a 60-node 6-hop network is 10 minutes.

The FTSP time-stamping reduces non-determinism to such a degree that RBS's receiver-receiver and TPSN's two-way sender-receiver synchronization become unnecessary and one-way sender-receiver synchronization suffices. FTSP accomplishes this by identifying, estimating, and compensating for almost all sources of uncertainties in the radio message delivery. We fully leverage and extend FTSP's results to build our time-stamping primitive presented in detail in Section 4.4.

Hardware accelerators represent the most promising approach to reducing jitter uncertainty to the level of the system clock granularity. Hill and Culler (2002) observed that unlike wide-area time synchronization protocols such as NTP, in sensor networks we may be able to determine all sources of communication delay. By exposing all sources of delay to the application, they were able to synchronize a pair of nodes to within 2μs of each other using sender-receiver time-stamping. The sources of jitter included several factors: raw RF transmission jitter of $\pm 1\mu$s on the transmission propagation due to the internals of the radio, hardware capture of the arriving pulse to within an accuracy of $\pm 0.25\mu$s, and synchronization code path delay of an additional $\pm 0.625\mu$s. This implementation was only possible because of the rich interface between the radio hardware and system software.

Dutta (2004) advocates the use of reactive timesync algorithms for random event detection, arguing that reduced power consumption and low latency due to the lower messaging frequency and no need for calibration, respectively, map well to the needs of event detection. Xu et al. (2004) used an approach similar to ours for time-stamping data in a WSN for structural monitoring. Both of these works focused on specific applications and neither generalized their approach to be broadly applicable.

Canonical Service	Typical Applications	APIs

Event time-stamping	Intrusion detection
	Countersniper system
	Source localization
		`SendToSink`
		`SendToAll`
		`OnReceive`

Data series time-stamping	Habitat monitoring
	Structural monitoring
	Environment monitoring
	Volcano event monitoring
	Target classification
	Beamforming
		`SetAccuracy`
		`MarkStart`
		`MarkStop`

Virtual global time	Debugging traces	`GetGlobalTime`
		`LocalToGlobal`
		`GlobalToLocal`

Coordinated action	Scheduled data collection Communications scheduling	`Schedule` `OnSchedule`

Table 1: Canonical time synchronization services, their typical applications, and proposed API calls and events that can be used in a general implementation. Events are callbacks from a service layer to a higher layer and are of the form `OnXXX`.

3 Canonical Services

Many sensor network applications require time synchronization but the *model* of synchronization that is needed varies greatly. In practice, many application developers resort to the virtual global time service without considering other alternatives. Virtual global time is chosen because it is conceptually simple but since it is proactive and message-intensive, virtual global time may use system resources unnecessarily. We argue that abstracting common timesync usage patterns from a number of existing applications and standardizing the interfaces of these abstractions will help developers better identify the synchronization needs of sensor network applications and help them choose the most efficient model for their needs.

This section presents a set of canonical time synchronization services derived from several well-known and common sensor network applications. Our focus is on the use of the services in *actual applications*, so we also propose a set of application programming interfaces (APIs) that could be used by these or other similar applications to access timesync services. However, this list is neither complete nor predictive: we did not abstract every single use of timesync found in the literature nor did we attempt to identify future models of synchronization. Nevertheless, this list of timesync services, each of which can be implemented using the proposed ETA primitive, suggests that other protocols, not on the list, could be implemented using ETA as well.

Table summarizes the canonical services, APIs and typical applications. Interestingly, we tried hard to find an application that truly required global time synchronization. This indicates a possible mismatch between the areas which have garnered much of the community's attention and the actual needs of common WSN applications.

The remainder of this section delves into the canonical services and their underlying abstractions, APIs, and motivating applications in greater detail. For the remainder of this paper, we restrict the discussion to only those applications which require temporal order, since logical and delivery orders are trivially implemented with messaging and causal ordering requires a temporal ordering capability.

3.1 Event Time-stamping

A single, isolated event is observed and time-stamped by one or more nodes at possibly different times. The observation from each node is to be collected at one or more sink nodes, either individually or in aggregate. The sink node must determine the time-stamp of each event in the sink's local time, *regardless of whether the sink is coordinated with an external source of time like GPS*. This service extends to multiple events through repeated invocation.

An event time-stamping API that consists of a `Packet` data structure augmented with an `eventTime` field, along with calls to convergecast a packet (i.e. send a packet toward the sink node)

```
command bool SendToSink(Packet packet)
```

flood a packet (network-wide) to all other nodes

```
command bool SendToAll(Packet packet)
```

and signal an event when new packet is available at a node (which is signaled at every intermediate hop)

```
event void OnReceive(Packet *packet)
```

appears sufficient to meet the timesync need of these applications.

3.2 Data Series Time-stamping

One or more observer nodes are periodically sampling some phenomena. The data series from each node is to be collected at a sink node for correlated analysis. Data series time-stamping requires time-stamps on each sample that are both accurate with regards to a series (intra-series accuracy) as well as accurate across series (inter-series accuracy). Unlike single event time-stamping, which is a discrete event and can benefit from discrete corrections such as a single RBS exchange, data series time-stamping is a continuous effort that requires continuous maintenance.

The precision of the data series time-stamps varies across applications. The phenomena in habitat monitoring are at a slow enough time scale that delivery order at a base station is a reasonable form of time-stamping. In contrast, the phenomena in structural health monitoring require in-network stamping to within a fraction of a packet transmission time. At the extreme are the applications in which the precise phase difference of a signal arriving at multiple nodes is important. For example, acoustic source localization and beamforming applications require signal phase be measured to within a fraction of a cycle, which for an acoustic signal in the kHz range might be just a few tens of microseconds. In many signal processing algorithms, samples cannot be dropped to compensate for clock errors. The challenge is how to compare observations when the sampling frequencies of the sources are not identical, the observations are collected over a (potentially long) period of time, and samples cannot be dropped or added (i.e. time must not have any sudden jumps).

A data series time-stamping API that includes calls to set the required inter-series accuracy

```
command bool SetAccuracy(int micros)
```

indicate the beginning of a data series (which initiates any under-the-hood work needed to meet the specified accuracy)

```
command bool MarkStart()
```

and indicate the end of a data series (which ends any under-the-hood work started to meet the specified accuracy)

```
command bool MarkStop()
```

could meet the timesync need of these applications provided that the underlying implementation could meet the constraints. Each timesync implementation might have its own method for satisfying the accuracy requirements of the application and no implementation might be able to satisfy all constraints.

3.3 Virtual Global Time

Virtual global time is perhaps conceptually the simplest of the time synchronization services to understand but is often the one which requires the most overhead to achieve. The basic idea is simple: a single, virtual, and globally shared timebase is accessible at each node in the network. An important question is where the timebase originates. Options include an external source like GPS, a distinguished root node, or some kind of network statistic as the network mean.

A virtual global time API that includes calls to read the current global time

```
command bool GetGlobalTime(int *time)
```

and to convert between the local times and their corresponding global times

```
command bool LocalToGlobal(int *time)
command bool GlobalToLocal(int *time)
```

appears sufficient to meet the timesync need of these applications.

3.4 Coordinated Action

Coordinated action is similar to event time-stamping, except that the event is in the future. Applications that require coordinate action include scheduled data collection and structural health monitoring. In the former, the action is a discrete data collection: depending on the different required fidelities, the synchronization needed for coordinated action can be used for the events themselves. In the latter, the action is a data time series; although the coordination can satisfy the inter-node accuracy, the data series service must still be responsible for intra-node accuracy.

The challenge in coordinated action is that, unlike simple time-stamping, it cannot benefit from post-facto synchronization such as an RBS exchange. However, unlike virtual global time, the nodes do not need a continuous synchronized time value: they merely need to agree on a single point in time. While an implementation of this service can depend on virtual global time, other implementation options exist.

We propose a coordinated action API that includes calls to schedule an action, in a node's *local* time, with optional quality of service parameters and number of repetitions

```
command bool Schedule(int *time)
command bool Schedule(int *time, int accuracy)
command bool Schedule(int *time, int accuracy,
                      int repeat)
```

and signal an event when it is time to act, for the `count`-th time, on some previously received coordinated action request

```
event void OnSchedule(int count)
```

---

4 Sources of Time Synchronization Error

This section establishes the terminology and identifies the various sources of error that contribute to time

synchronization uncertainty. We analyze two fundamentally different types of error: 1) time-stamping jitter encountered when transmitting radio messages between nodes, and 2) time errors caused by the manufacturing and operating differences of physical clocks. We also provide guidelines on how to design and implement high precision one-way sender-receiver timestamping service for broadcasted messages.

4.1 Time-stamping Jitter

The sources of error in message time-stamping need to be carefully analyzed and compensated for because nondeterministic delays in radio message delivery can be magnitudes larger than the required precision of time synchronization. We use the following decomposition of these errors first introduced by Kopetz and Ochsenreiter (1987) and later extended by Kopetz and Schwabl (1989); Horauer et al. (2002) and Ganeriwal et al. (2003). The presented list is a generalization of a similar list presented by Maróti et al. (2004).

1. *Send Time*—the time used to assemble the message and issue the send request to the MAC layer on the transmitter side. This depends on the system call overhead and processor load.

2. *Access Time*—the delay incurred waiting for access to the transmit channel up to the point when transmission begins. This depends on the channel contention.

3. *Transmission Time*—the time required for the sender to transmit the message. This depends on the length of the message and the speed of transmission.

4. *Propagation Time*—the time required for the message to propagate from the sender to the receiver once it has left the sender. This depends only on the distance between the two nodes.

5. *Reception Time*—the time required for the receiver to receive the message. The transmission and reception times have similar characteristics and overlap, if the propagation time is negligible.

6. *Receive Time*—time to process the incoming message and to notify the receiver application. Its characteristics are similar to that of the send time.

We further analyze the sources of uncertainties in the overlapping transmission and reception times by observing an idealized point of radio message, such as the end of a particular byte. We follow the transmission of this idealized point through the software, hardware and physical layers of the wireless channel from the sender to the receiver. The following delays in the propagation of the idealized point seem to be the most important:

7. *Interrupt Handling Time*—the delay between the radio chip raising and the microcontroller responding to an interrupt by recording a time-stamp.

8. *Encoding Time*—the time required for the radio chip to encode and transform a part of the message to electromagnetic waves. This starts when the radio chip initiates the transfer of the idealized point.

9. *Decoding Time*—the time required for the radio chip on the receiver side to transform and decode the message from electromagnetic waves to binary data. This ends when the radio chip raises an interrupt indicating the reception of the idealized point.

Some radio chips cannot capture the byte alignment of the transmitted message stream on the receiver side and the radio stack has to determine the bit offset of the message from the alignment of a known synchronization byte. Since the transmission time of a byte is a few hundred microseconds at 38.4 kbps, the delay caused by the incorrect byte alignment must be compensated for.

10. *Byte Alignment Error*—the delay incurred because of the different byte or data segment alignment of the sender and receiver. This is deterministic and can be computed at the receiver from the bit offset and the speed of the radio.

The list provided above characterizes phenomena relevant to the time-stamping of messages transmitted over the wireless channel. We direct the reader to a paper by Maróti et al. (2004) for a more comprehensive analysis of these sources on the Mica2 platform. Note that some radio technologies are not impacted by all these sources of error.

4.2 Physical Clocks

Computer systems typically measure time in discrete steps by counting the oscillations of a physical clock. Such clocks are driven by quartz crystals, ceramic resonators, or resistor-capacitor oscillator circuits, depending on the accuracy, stability, power, startup time, and cost requirements of the applications. More advanced solutions are available as ultra-stable oscillators constructed from temperature-compensated sapphire resonators. However, for majority applications, common quartz crystals provide a reasonably accurate and cost-effective solution. We assume that a quartz crystal is the basis of the physical clock for the remainder of this section and use the terms crystal and physical clock interchangeably.

A quartz crystal oscillator is an analog component which outputs a continuous-time sinusoidal signal converted to a digital signal by microcontroller. The period of the physical clock sets the length of the discrete time quantum, or *granularity*, of the timebase. All synchronization protocols encounter errors caused by the insufficient precision of the time representation when an analog-digital boundary is crossed (like event detection or time-stamping). The exception to this is the time-triggered technology that can phase synchronize the clocks across the distributed system.

11. *Time Representation Error*—the delay between an event and the nearest (or next) representable time value.

Under ideal circumstances, physical clocks oscillate at a constant frequency. In the real world, manufacturing variations and exposure to out-of-tolerance conditions (e.g. mechanical shock) result in permanent frequency errors of crystals and variations in temperature and age result in short-term errors of the crystals.

12. *Frequency Skew*—under specified conditions at a nominal ambient temperature, the difference between the crystal frequency and the nominal frequency. Usually expressed as $\Delta f/f$ with typical values around ± 50 ppm.

13. *Temperature Characteristics*—taking the frequency at a nominal ambient temperature as the reference (usually $+25°$C), the change in frequency with respect to a change in the ambient temperature. Different crystal types have different temperature characteristics:

    - For tuning-fork crystals, $\Delta f/f$ depends quadratically on the difference between the ambient and the nominal temperature.
    - For AT-cut crystals, the relationship between $\Delta f/f$ and the temperature difference is described by a 3^{rd} order polynomial equation.

14. *Aging*—amount of frequency drift when operated under the specified conditions for a specified term. A typical value for aging might be -5 ppm in the first year.

4.3 Minimizing the time-stamping jitter

The uncertainties of the send, receive, access and byte alignment times are best eliminated by time-stamping the message in the MAC-layer, as is done in several time synchronization protocols. The propagation time cannot be calculated or compensated for within the transmission of a single message. However, in static networks it has no jitter and over short distances (less than 300 meters) its duration is negligible (less than one microsecond). We argue that the message needs to be time-stamped at an idealized point, practically when an interrupt is raised by the radio. This eliminates the effect of the overlapping transmission and reception times. The jitter in the interrupt handling time is best eliminated by utilizing a capture register on the microcontroller. If the time-stamp is taken in software, it is important to minimize the length of all atomic sections in the application to minimize the amount of time interrupts are disabled. The encoding time lasts several hundreds of microseconds but usually has very low jitter. On the other hand, the radio technology, signal strength fluctuations and bit synchronization errors will introduce jitter in the decoding time. The following novel technique can be used to reduce the jitter of the encoding, decoding and interrupt handling times. We record multiple time-stamps both at the sender and receiver sides as the message is being transmitted, then using statistical analysis we arrive at the final time-stamps (one at the sender and one at the receive side) that has lower jitter than the individual ones. Details of the statistical analysis is presented by Maróti et al. (2004).

The latest generation of radios make the processor-radio interface richer without burdening the processor with the overhead of directly coordinating the transmission. Such improvements also enable fine-grained timing information to be communicated from the radio to the processor while using an abstract and high-level interface. For example, the Chipcon CC2420 radio, which supports the IEEE 802.15.4 standard, allows random access into the transmit FIFO *during* transmission (Chipcon, 2003). Such access to the transmit queue, when coupled with the Start of Frame Delimiter (SFD) output signal—a deterministic hardware interrupt on the CC2420—enables fine-grained message time-stamping with virtually no jitter uncertainty on the transmit end and $\pm 0.125\mu$s of uncertainty on the receive end. By connecting the SFD signal to a timer capture pin on the processor, accurate time-stamping on the order of the time representation error is possible.

The presented techniques reduce only the jitter of these times, but do not estimate their absolute value. The sum of these delays is the difference between the sender and receiver side time-stamps. This sum can be measured with a one-time calibration procedure utilizing a many-to-many message handshake similar to the RBS algorithm. We point out that using software tools only, it is impossible to measure the absolute value of the encoding and decoding times separately, only their sum.

We direct the reader to the paper by Maróti et al. (2004) for a more comprehensive analysis of these techniques on the Mica2 platform in which 1.4 microsecond time-stamping precision was reported. Our key observations is that with deterministic MAC-layer time-stamping, we can eliminate the runtime calibration phase that is required in the work of Romer (2004) and Elson (2002), the need for a "third party" node as suggested by Elson (2002) and the Single-Pulse Synchronization presented by Elson (2003) for *many* of the existing applications of sensor networks.

4.4 The choice of local time

The first key design choice in any time synchronization protocol is the source of time that we want to synchronize. We argue that each node must have a free running local clock whose offset and skew is only measured and compensated for, but never altered. This local clock gives a natural time line where events are recorded, actions are scheduled and calculations are performed.

The time representation error of events can never be eliminated, but it can be reduced by using a higher frequency crystal. For example, the Mica2 mote includes a 32.768 kHz crystal which provides 30.52μs granularity which is sufficient for many applications. If a finer granularity clock is needed, the Mica2 also includes a 7.3728 MHz crystal which provides 0.1356μs granularity.

The actual frequency of a particular crystal instance can

take any value within the range specified by the tolerance precision. If a particular crystal's frequency is measured at multiple temperatures, we can create a calibration table that provides a frequency skew factor for every ambient temperature. Typically, this kind of calibration is performed at manufacture time for instrument-grade electronics like oscilloscopes and counters. The calibration must be repeated periodically because aging and shock can cause frequency variations. For WSNs that contain hundreds or thousands of nodes, repeated manual calibration is simply not an option. Instead, it might be possible to perform distributed parameter calibration, using the fact that sensor nodes have multiple crystals with different temperature characteristic curves. For example, on the Mica2 motes, the 32 kHz clock is based on a tuning fork crystal while the 7 MHz clock is an AT-cut crystal.

5 Implementation of Canonical Services

We argue that both the development and use of timesync services will be greatly enhanced by shielding developers from the myriad sources of time-stamping error presented in Section 3.4 by encapsulating these errors in a timesync primitive. Our reasons are as follow:

- The implementation of a timesync primitive is best done along with the analysis of the sources of error by the hardware domain experts. Consequently, the primitive can provide fine-tuned performance by reducing or eliminating nearly all sources of timesync errors.

- The timesync primitive is well suited to the component based architecture of distributed applications and can simplify the implementation of timesync services.

- The timesync primitive improves the portability of timesync services to a different hardware platform by isolating code changes to the timesync primitive.

- Finally, the timesync primitive can be implemented as a service provided by the operating system, making it easier to access hardware interfaces directly and enable higher precision time synchronization protocols to be implemented (Ganeriwal et al., 2003; Maróti et al., 2004) than would be possible otherwise.

In the remaining part of this section, we introduce our time-stamping primitive and show how it can be used to implement two of the canonical services described in Section . We demonstrate the significant decrease in complexity of the implementation of these services when compared to the existing protocols found in the literature.

5.1 The Elapsed Time on Arrival Time-stamping Primitive

Conceptually, the Elapsed Time on Arrival (ETA) algorithm is useful in the case when a sensor node detects a certain phenomenon of interest, or *an event*, and a neighbor of the sensor node needs to know the time of this event. ETA allows neighboring nodes to establish a common time base by sending a single radio message transmission. In our approach, the times of all events of interest are recorded in the local time of a node and inter-node time conversions occur whenever messages are transmitted from one node to another. At transmission time, the elapsed time since the event occurrence is computed and included as a separate field in the message. On receipt, a node computes the event time by subtracting the elapsed time from the receiving node's local time.

The implementation details for reducing timesync uncertainty for the Mica2 platform were discussed in Section 4.3 and can be also found in the work by Maróti et al. (2004). The pseudo code of the ETA algorithm is shown in Figure 1.

Each node accesses its local clock or timer through the `GetLocalTime` call. On an event occurrence, the local time is recorded in the `eventTime` field and the event message is enqueued for the transmission. When the event message is scheduled and the idealized point of the message is being transmitted (ostensibly after some delay in the media access control layer), both the sender and the receivers record their current local time into the `transmissionTime` field. The sender also updates the `eventTime` field to contain the elapsed time since the event occurrence. Note, that the `transmissionTime` is not transmitted over the wireless channel to the neighbors. After the whole message has been received, the `eventTime` is reconstructed on the receiver side by subtracting the elapsed time from the `receptionTime`. The key enabler of ETA is the ability to update the content of the radio message (to update the `eventTime` field) while the message is being transmitted.

The proposed time-stamping protocol eliminates or reduces all sources of time-stamping error except for propagation time and the effects of the clock skew between the sender and receiver while a message is waiting in the queue. Neither of these errors can be estimated with a single radio message and hence this estimation is left for higher layers to implement. The overhead of the algorithm is a single field in the transmitted message. After message transmission is complete, the sender and all receivers know the time of the transmission, as well as the time of the event in each receiver's local time base.

5.2 Event Time-stamping Service

The Routing Integrated Time Synchronization (RITS), a reactive time synchronization protocol, can be used to obtain event detections at multiple observers in terms of local time of the sink to within microseconds. RITS is an extension of time-stamping primitive ETA over multiple hops. It

```
struct TimeStampedMessage {
    char[] payload;
    int eventTime;     // elapsed time (msg in transmit)
                       // local time of event (otherwise)
    int transmissionTime; // not actually transmitted
}

OnBeginTransmission(TimeStampedMessage msg) {
    msg.transmissionTime = GetLocalTime();
    msg.eventTime = msg.transmissionTime - msg.eventTime;
}

OnBeginReception(TimeStampedMessage msg) {
    msg.receptionTime = GetLocalTime();
}

OnEndReception(TimeStampedMessage msg) {
    msg.eventTime = msg.receptionTime - msg.elapsedTime;
}
```

Figure 1: The Elapsed Time on Arrival (ETA) algorithm.

is achieved by integrating ETA into the multi-hop routing engine DFRF presented by Maróti (2004).

After detecting an event, the observer stores the relevant data in a local data structure along with the local time of the event occurrence and uses one of the methods of RITS to send the data to the sink. RITS, being a routing engine, sends the packet to the sink along a multi-hop path while maintaining the event time using ETA. At each hop it converts the event time from the local time of the sender to the local time of the receiver.

The details of our implementation are shown in Figure 2. Upon receiving a request to send a data packet, RITS creates a `TimeStampedRITSMessage` and off-loads all the local time conversion details to the ETA algorithm. Upon receiving a `TimeStampedRITSMessage`, RITS creates a new packet and associates with it `message.eventTime` which is the event time converted by ETA to the local time of the receiver. RITS then signals the reception of packet to the application level and possibly enqueues the packet for the further transmission.

5.3 Virtual Global Time Service

Rapid Time Synchronization (RATS), a proactive time synchronization protocol, achieves network-wide timesync in medium size networks with microsecond precision and rapid convergence. The basic idea is the same as in RITS, except that the direction of messages is reversed: instead of convergecast from multiple nodes to the sink, we use a broadcast from a single node, the root, to all other nodes in the network.

Time synchronization is initiated by the root by broadcasting a message containing two fields, `rootEventTime` and `eventTime`, which are both initialized to the same value in the root's local time (the current time). Network-wide broadcast is used to transmit the root's message to all nodes in the network. The `rootEventTime` field is not modified during the broadcast, while ETA converts the `eventTime` field to the local time of the receiver of the message. Thus when a node receives the message, it obtains two times corresponding to the same time instant: `rootEventTime`, in the local time of the root, and `eventTime`, in the local time of the node. We call the pair (`rootEventTime`,`eventTime`) a synchronization point and use several of them to estimate the offset and skew of the local clock of receiver to the root's clock. Consequently, the local time of the root becomes the global time in the network.

The most widely used method to estimate the clock drift of a node to the clock of the root is linear regression (Maróti et al., 2004; Elson, 2002): a fixed number of synchronization points are locally stored in a table at each node, a linear regression line is fit to these points and the slope and intercept of the line is used to estimate the root's time in the future. Since the drift of the clock may change over time (see Section 3.4), the root needs to rebroadcast synchronization messages periodically. We use a sequence number field in the RATS message to distinguish new synchronization messages from old ones. We used RITS to implement RATS (see Figure 3) but it is straightforward to implement RATS directly over ETA.

6 Evaluation

We implemented the ETA, RITS and RATS protocols in TinyOS (Hill, 2000) on the UC Berkeley Mica2 platform. Even though the RITS and RATS protocols are simple extensions of ETA, our experimental evaluation shows that

```
struct RITS_Packet {
    char[] data;
    int    eventTime;     // not actually transmitted
}

SendToSink(RITS_Packet packet) {
    TimeStampedRITSMessage msg;
    msg.payload  = packet.data;
    msg.eventTime = packet.eventTime;
    MessageSend(msg);    // transmit over the radio
}

OnMessageReceive(TimeStampedRITSMessage msg) {
    RITS_Packet packet = new RITS_Packet();
    packet.data = msg.payload;
    packet.eventTime = msg.eventTime;
    signal OnReceive(&packet);
    Enqueue(packet);    // transmit via routing
}
```

Figure 2: Pseudo code of the RITS implementation.

```
struct RITS_Packet {
    int    rootEventTime;
    int    eventTime;
}
OnRootSend() {
    RITS_Packet packet;
    packet.eventTime = GetLocalTime();
    packet.rootEventTime = packet.eventTime;
    RITS.SendToAll(packet);
}

event RITS.OnReceive(RITS_Packet *packet) {
    LinRegression (packet->rootEventTime, packet->eventTime);
}
```

Figure 3: Pseudo code of the RATS implementation.

their performance is comparable to, if not better than, the leading timesync algorithms found in the WSN literature.

6.1 Evaluation of RITS - Routing Integrated Time Synchronization

We tested the performance of RITS in an experimental setup using 45 Mica2 motes arranged in a grid. Each node was able to communicate only with its neighbors as shown in Figure 4, forming a 10-hop network. This constraint was enforced in software. The test scenario involved multiple observers detecting the same event and reporting the event detection times to the sink. We simulated the event detection by a radio message that was transmitted by a single node and received by all the nodes that were at most 2 hops away from the sender. Figure 4 shows the sink, the nodes, the radio links and the detection radius of an event.

Each node recorded the time of arrival of the message as an event detection time (`message.receptionTime` in Figure 1), created a detection message consisting of the id of the node (`message.nodeID`) and the detection time (`message.eventTime`) and sent this message to the sink. Consequently, the sink received detection messages from up to 12 different nodes. Each such message contained the time of the simulated event at a particular node converted by RITS to the local time of the sink. The times reported by two different nodes should be the same if we neglect the propagation time of a radio message. However, the time-stamping error as well as the clock skew error cause a small variance of the received times. For each group of n reported times we calculate the maximum and the average of the $\binom{n}{2}$ pairwise errors in the group. We call

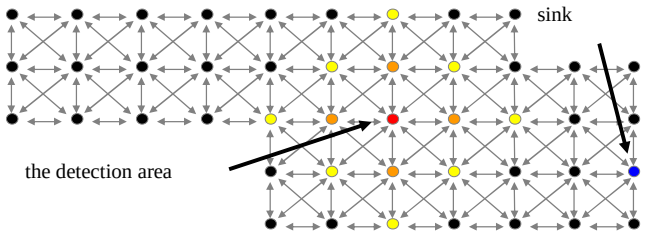

Figure 4: The layout of the RITS experiment: a node communicates with at most 8 nodes and an event is detected at most at 12 nodes.

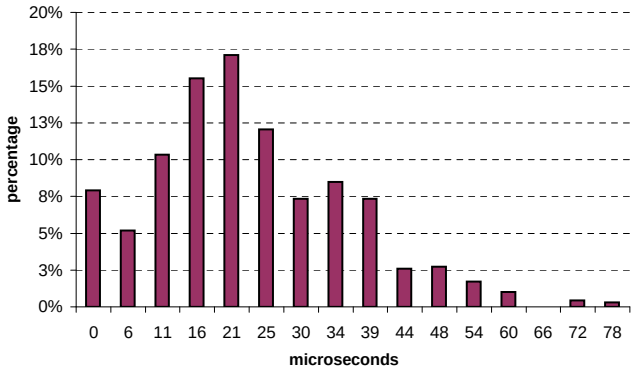

Figure 5: Shows the histogram of the *maximum* pairwise error in the RITS experiment.

these values the *average* and *maximum* synchronization error, respectively.

We broadcasted the event simulation message 5 times with 100ms delays in a single simulation round. The sender of the event simulation message was chosen randomly both within and across rounds. We initiated consequent simulation rounds with a 30-second period, let the experiment run for one and one-half hours and collected data for 900 simulated events. We plot the histogram of *maximum* synchronization error in Figure 5. The maximum and average error of the maximum synchronization error over all rounds were 80.19μs and 7.86μs, respectively.

The performance of RITS is principally affected by two factors. First, variations in clock crystal frequencies—we have observed delays of up to 17 seconds between the event happening and the event notification message arriving to the sink in a particular round. The frequency error of a typical oscillator is in the range of 10ppm-100ppm, therefore, 7MHz MICA2 oscillator can introduce significant errors over these long delays. Second, variations in the jitter of time-stamping – this affects RITS the same way as any other timesync protocol that uses radio message time-stamping to establish a common time base.

The timesync accuracy of RITS as shown by our experiment is slightly worse than the accuracy of published proactive algorithms by Ganeriwal et al. (2003); Maróti et al. (2004); Elson (2002). However, we argue that this accuracy is sufficient for most applications and that the energy-quality tradeoff of RITS vs. proactive protocols is acceptable, and even desirable, in many cases.

The benefits of RITS over proactive timesync protocols are as follows. First, the drifts of the local clocks of the nodes do not need to be calculated and maintained in RITS whereas proactive algorithms require periodic resynchronization. Second, RITS can be powered down completely and woken up by the event itself (as in Lédeczi et al. (2005)), which enables more sophisticated power management techniques than proactive protocols which require the nodes to be continuously active, even between synchronization events, since at least local clocks need to be running. Third, RITS introduces minimal communication overhead; there is no need for time synchronization related messages as in proactive algorithms and only a single field needs to be included in the transmitted message.

6.2 Evaluation of RATS - Rapid Time Synchronization

We evaluated the performance of RATS using the same test setup as was used for FTSP (Maróti et al., 2004). We set up 60 Mica2 motes in a 5x12 grid so that each node had up to 8 neighbors enforced in software. This is similar to the RITS test scenario shown in Figure 4. The test setup involves 60 synchronizing nodes, the reference broadcaster that queries the global time estimates of the synchronizing nodes, and the base station that collects the reported global times. The reference broadcaster and the base station are used just for evaluation purposes, they play no role in the synchronization algorithm.

The root was programmed to transmit synchronization messages first for 10 seconds with a period of 2 seconds and then for the rest of the experiment with a period of 30 seconds. The reference broadcaster queried the global time from all the nodes with a period of 5 seconds during the first 2 minutes and with a period of 23 seconds (which is a relative prime to 30) the rest of the time. In each reference broadcast we obtained up to 60 reported global times, one of them being the root's time. For each reference broadcast, we computed the *maximum* and the *average* absolute errors as the maximum and the average absolute difference between the root's time and the other times for each reference broadcast round.

We ran the experiment for 6 hours and achieved network-wide synchronization (the point in time when at least two synchronization points had been received by all nodes) only 4 seconds after switching on the root. Over all rounds, the maximum and average errors of RATS were 26μs and 2.7μs, respectively. This is in contrast to the 10-minute convergence time, 14μs maximum error and 2.3μs average error of FTSP (Maróti et al., 2004). We note that the FTSP experiment had only 6 hops whereas our experiment had 11 hops, so we suspect that the maximum and average errors of RATS and FTSP might have been closer if FTSP were synchronizing over 11 hops. Figure 6 shows the maximum and average errors for each round of the experiment.

At the beginning of experiment the errors were generally

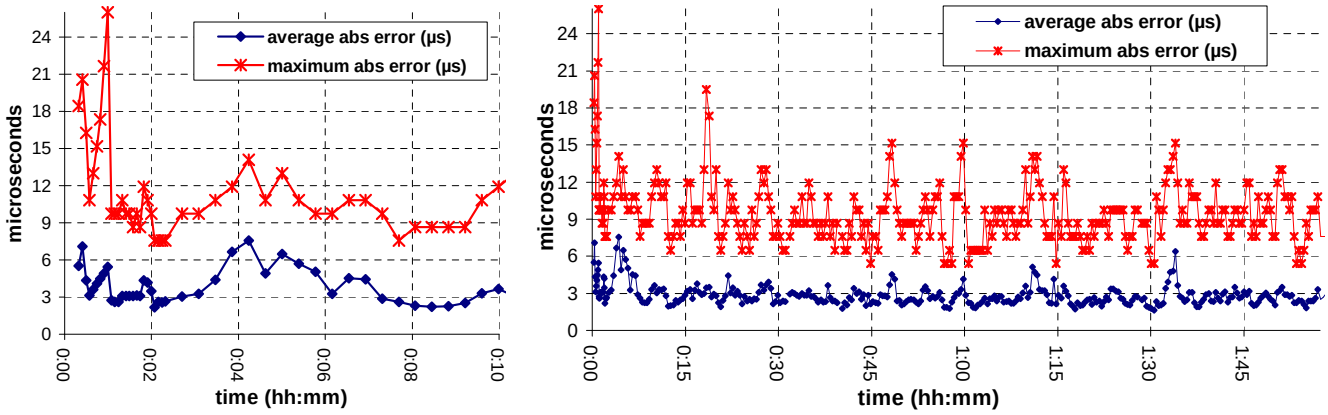

Figure 6: The RATS experiment showing the maximum and average errors of the reported global times. The figure on the left shows the first 10 minutes and the figure on the right shows the whole experiment. The synchronization was achieved after 4 seconds and the average error was 2.7μs in the 11 hop network.

larger than later on. This is expected as initially only a few datapoints were available in the regression table and the clock skew estimate was not as accurate. Moreover, errors in individual measurements have much larger effect on the accuracy of the skew estimation, because initially the regression table covers only a few seconds. The data also illustrate this point: when the protocol switched to the 30-second period, the maximum error gradually increased to 26μs and then gradually decreased as the skew estimates improved. More generally, the tradeoff between RATS and FTSP are as follow.

Network-wide convergence time of timesync in RATS was shown to be 4 seconds for 11 hops; it took 10 minutes to achieve network-wide synchronization of FTSP for a 6-hop network as reported by Maróti et al. (2004). Depending on the accuracy needs of the application it is possible to achieve sub-second convergence using RATS by decreasing the initial synchronization period even further.

The robustness of RATS is superior to FTSP in two regards. First, the synchronization error of a node in RATS does not depend on the current clock drift estimation of the nodes on the route from the root but it does for FTSP. Second, the synchronization message from the root arrives at a each node along multiple routes, giving the node multiple local times for each global time. RATS takes the *median* of the received local times as its synchronization point, which is robust to outliers, and allows RATS to tolerate faulty or adversarial nodes whereas FTSP cannot.

Unlike FTSP, RATS cannot maintain network-wide synchronization if the network is partitioned and is vulnerable to a root failure. However, since most WSN applications depend on a gateway for wide-area network access, we have the base station already. Since RATS floods the network with radio messages during synchronization times, the available bandwidth of the network is temporarily but dramatically reduced. FTSP, in contrast, distributes the radio message load evenly, thus it provides constant radio bandwidth for applications.

7 Conclusions

Advances over the last few years in radio and wireless sensor network design enable, and increasingly challenging application requirements drive us to continuously reconsider the boundaries between the hardware, operating system, middleware and application layers. Single layer solutions to several key problems that inherently touch multiple layers, such as time synchronization in wireless networks, are likely to be less efficient than those that can carefully address the challenges at each layer. The price one must pay for the increased control and efficiency, however, is additional complexity between the layers. We argue that the solution is to minimize the number of different interfaces between the layers but increase their expressive power.

Our first key contribution is the careful analysis of the sources of time synchronization jitter between the operating system, radio chip and wireless channel. We present a comprehensive list of software techniques and recommended hardware solutions to maximize the precision of the message time-stamping. Then we propose a time synchronization primitive, called Elapsed Time on Arrival (ETA) which is a one-way, sender-receiver time-stamping service for broadcasted messages. On the Mica2 platform its precision is better than 1.4μs between the sender and/or any of the receivers. It allows the translation of event times, expressed in the local clock of the nodes, from the sender to its neighbors. Other than the event time, it has absolutely no data overhead in the message. The two key enablers of ETA are 1) integration with the MAC-layer to minimize the jitter, and 2) the ability to embed the time-stamp in the message while it is being transmitted.

As our second contribution, we have identified a small set of canonical services together with their application programming interfaces that capture the time synchronization requirements of actual wireless sensor network applications. We argue that abstracting common timesync usage patterns from the existing applications will help ap-

plication developers to identify their time synchronization needs. We propose to build canonical services on top of an explicit timesync primitive, such as our ETA, and show that their implementation with ETA is less complex and has comparable or better precision. The Event Time-Stamping canonical service is implemented by the Routing Integrated Time Synchronization (RITS) protocol, which is the multi-hop extension of ETA. Combined with a convergecast routing policy, it allows the root node to correlate the detection time of a single event at multiple observers. In a 45-node 10-hop network the maximum and average time synchronization errors were 80μs and 8μs, respectively. The Virtual Global Time canonical service was implemented by the Rapid Time Synchronization (RATS) protocol, in which a root node broadcasts its local time in the network using RITS. In a 60-node 10-hop network the algorithm achieved network-wide synchronization in 4 seconds, and the maximum and average time synchronization errors were 26μs and 2.7μs, respectively. In a similar setup, the Flooding Time Synchronization Protocol (FTSP) achieved network-wide synchronization in 10 minutes, while the maximum and average errors were 14μs and 2.3μs, respectively.

Acknowledgements

The DARPA/IXO NEST program has partially supported the research described in this paper.

REFERENCES

Blum, P., Meier, L., and Thiele, L. (2004). Improved interval-based clock synchronization in sensor networks. *Third International Symposium on Information Processing in Sensor Networks*, pages 349–358.

Chipcon (2003). *Chipcon CC2420 Datasheet: 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver (v 1.2)*.

Dai, H. and Han, R. (1994). Tsync: A lightweight bidirectional time synchronization service for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 8(1):125–139.

Dutta, P. K. (2004). On random event detection in wireless sensor networks. Master's thesis, The Ohio State University.

Elson, J. (2002). Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*.

Elson, J. (2003). *Time Synchronization in Wireless Sensor Networks*. PhD thesis, University of California, Los Angeles.

Elson, J. and Romer, K. (2002). Wireless sensor networks: A new regime for time synchronization. In *Proceedings of the First Workshop on Hot Topics in Networks (HotNets-I)*.

Ganeriwal, S., Kumar, R., and Srivastava, M. B. (2003). Timing-sync protocol for sensor networks. In *ACM SenSys 2003*, Los Angeles, California, USA.

Hill, J. (2000). A software architecture supporting networked sensors. *Master's thesis, U.C. Berkeley Dept. of Electrical Engineering and Computer Sciences*.

Hill, J. and Culler, D. (2002). Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24.

Horauer, M., Schossmaier, K., Schmid, U., Höller, R., and Kerö, N. (2002). Evaluation of a high precision time synchronization prototype system for ethernet lans. *34th Annual Precise Time and Time Interval Meeting (PTTI)*.

K. Römer, P. Blum, L. M. (2005). *Time Synchronization and Calibration in Wireless Sensor Networks, To appear in: I. Stojmenovic (Ed.), Wireless Sensor Networks*. Wiley and Sons.

Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.

Kopetz, H. and Ochsenreiter, W. (1987). Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–939.

Kopetz, H. and Schwabl, W. (1989). Global time in distributed real-time systems. Technical Report 15/89, Technische Universitat Wien, Austria.

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

Lédeczi, A., Völgyesi, P., Maróti, M., Simon, G., Balogh, G., Nádas, A., Kusý, B., Dóra, S., and Pap, G. (2005). Multiple simultaneous acoustic source localization in urban terrain. *In Proc. 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*.

Li, Q. and Rus, D. (2004). Global clock synchronization in sensor networks. *IEEE INFOCOM*.

Maróti, M. (2004). Directed flood-routing framework for wireless sensor networks. *In Proc. 5th ACM International Middleware Conference*, pages 99–114.

Maróti, M., Kusý, B., Simon, G., and Lédeczi, A. (2004). The flooding time synchronization protocol. *In Proc. 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 39–49.

Marzullo, K. and Owicki, S. (1983). Maintaining the time in a distributed system. *Second annual ACM symposium on Principles of Distributed Computing*, pages 295–305.

Meier, L., Blum, P., and Thiele, L. (2004). Internal synchronization of drift-constraint clocks in ad-hoc sensor networks. *Fifth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 90–97.

Reichenbach, H. (1957). *The Philosophy of Space and Time*. New York: Dover.

Romer, K. (2004). Time synchronization in ad hoc networks. In *MobiHoc 2001*.

Schmid, U. and Schossmaier, K. (1997). Interval-based clock synchronization. *Real-Time Systems*, 12(2):172–228.

Sichitiu, M. L. and Veerarittiphan, C. (2003). Simple, accurate time synchronization for wireless sensor networks. *IEEE Wireless Communications and Networking Conference (WCNC03)*.

Sivrikaya, F. and Yener, B. (2004). Time synchronization in sensor networks: A survey. *IEEE Network*, 18(4):45–50.

Su, W. and Akyildiz, I. F. (2004). Time-diffusion synchronization protocol for sensor networks. *To appear in IEEE/ACM Transactions on Networking*.

van Greunen, J. and Rabaey, J. (2003). Lightweight time synchronization for sensor networks. *2nd ACM International Workshop onWireless Sensor Networks and Applications*, pages 11–19.

Xu, N., Rangwala, S., Chintalapudi, K., Ganesan, D., Broad, A., Govindan, R., and Estrin, D. (2004). A wireless sensor network for structural monitoring. *Proceedings of ACM Sensys 2004, Los Angeles, California*.

Biographical notes

Branislav Kusy is a Research Assistant at the Institute for Software Integrated Systems, Vanderbilt University. His current research interests include time synchronization, radio based ranging and localization of sensornets. He is working towards a PhD in the Electrical Engineering and Computer Science Department at Vanderbilt University, Nashville.

Prabal Dutta is a doctoral student in the Computer Science Division of the Electrical Engineering and Computer Sciences Department at the University of California, Berkeley, where he is an NSF Fellow advised by David Culler. His interests include sensor network design at the hardware software interface.

Philip Levis was a doctoral student in the Computer Science Division of the Electrical Engineering and Computer Sciences Department at the University of California, Berkeley, at the time of this work, where he was advised by David Culler. He is currently Assistant Professor of Computer Science at Stanford University. His research interests are sensor network systems, protocols and programming. He implemented TOSSIM, the de facto simulator of the sensor network community and helped design nesC, the TinyOS programming language.

Miklos Maroti was a Research Assistant Professor at the Institute of Software Integrated Systems, Vanderbilt University at the time of this work. Currently, he is Assistant Professor at University of Szeged, Hungary. His research interest includes formal specification and analysis of embedded systems, and active libraries of middleware components. He received a PhD in mathematics from Vanderbilt University.

Akos Ledeczi is a Senior Research Scientist at the Institute for Software Integrated Systems, Vanderbilt University. His current research interests include model-based synthesis and simulation of embedded systems. He received a PhD in electrical engineering from Vanderbilt University.

David Culler is Professor of Computer Science at University of California, Berkeley, a former Director of Intel Research Berkeley, and a member of the National Academy of Engineering. He leads the TinyOS project.