Implementation of an Adaptive Dissemination Protocol on Sensor Nodes

Student: Dmitry Lukyantsev

Semester Project

Start Date: 10. May 2010

End Date: 16. August 2010

Advisors: Federico Ferrari, Marco Zimmerling

Supervisor: Prof. Lothar Thiele

Swiss Federal Institute of Technology Zurich

Department of Information Technology and Electrical Engineering

Computer Engineering and Networks Laboratory

Abstract

Dissemination protocols are used to deliver messages (e.g. time-stamps, code, data) from any node (source node) to the rest of the network within a short time period. A new dissemination protocol for wireless sensor networks – Secondis – has recently been proposed. The protocol is designed to quickly and reliably disseminate very short time-stamped messages required by synchronization protocols. In this thesis, we implement Secondis using the Contiki OS features and evaluate it based on simulation experiments, using Tmote Sky as the target wireless sensor platform. We identify limitations imposed by the hardware and determine optimal parameters of the protocol.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Problem Statement

Wireless sensor networks (WSNs) are emerging systems for applications such as industrial automation, health-care, surveillance and safety monitoring [2]. WSNs may present real-time requirements, such as bounded end-to-end delay for the data collection. Many sources of non-determinism may however affect the timing behavior, most importantly the inherent unreliability of the wireless channel and possible collisions that occur when two or more sensor nodes simultaneously transmit to a common neighbor. In real-time WSN scenarios it is of fundamental importance to achieve highly predictable communication schemes to minimize interference.

Global time synchronization is an important requirement in real-time WSNs, where nodes have to take time-coordinated actions in order to provide the desired reliability. Interference-free communication schemes assume that nodes are synchronized; their predictability strongly depends on that of the underlying time synchronization protocol. Time synchronization is also required for consistent distributed sensing and control. Furthermore, common services in WSN, such as coordination, communication, security, power management or distributed logging also depend on the existence of global time.

Dissemination protocols are used to deliver time-stamped messages from a source node to the rest of the network within a short time period, avoiding simple broadcast retransmissions, that lead to the broadcast storm problem [5], and considering limited energy, computational power, and communication resources available to the sensors in the network.

This thesis aims to implement the Secondis dissemination protocol [1]. We develop software that implements protocol behavior using the Contiki OS [7]. We use COOJA [8] to simulate the protocol. The environment emulates Tmote Sky [11] wireless sensor module and allows us to debug and evaluate the code before implementing it on a real sensor.

1.2 Outline

We briefly describe time-synchronization and dissemination protocols in Chapter 2. Practical implementation and tools description is given in Chapter 3. In Chapter 4, we describe simulation experiments and analyze the results.

Chapter 2. Time - Synchronization and Dissemination Protocols in WSNs

Several time-synchronization and dissemination protocols for WSNs exist. Below we introduce most prominent of them and talk about new dissemination protocol – Secondis.

2.1 Time – synchronization protocols

Time synchronization algorithms provide mechanisms to synchronize the local clocks of the nodes in the network. The problem has been extensively studied in the past. The most widely adapted protocol used in the Internet domain is the Network Time Protocol (NTP). The NTP clients synchronize their clocks to the NTP time servers with accuracy in the order of milliseconds by statistical analysis of the round-trip time. The time servers are synchronized by external time sources, typically using GPS. The NTP has been widely deployed and proved to be effective, secure and robust in the Internet. In WSN, however, non-determinism in transmission time caused by the Media Access Channel (MAC) layer of the radio stack can introduce significant delay at each communication hop. Therefore, without further adaptation, NTP is suitable only for WSN applications with low precision demands and loose energy constraints, since the GPS module and the protocol's communication scheme require additional power.

2.1.1 Reference broadcast synchronization (RBS) algorithm and timing-sync protocol for sensor networks (TPSN)

Two of the most prominent examples of existing time synchronization protocols developed for the wireless sensor network domain are the RBS algorithm and the TPSN [2].

In the RBS, a reference message is broadcasted. The receivers record their local time when receiving the reference broadcast and exchange the recorded times with each other. The main advantage of RBS is that it eliminates transmitter-side non-determinism. The disadvantage of the approach is that additional message exchange is necessary to communicate the local time-stamps between the nodes.

The TPSN algorithm first creates a spanning tree of the network and then performs pairwise synchronization along the edges. Each node gets synchronized by exchanging two synchronization messages with its reference node one level higher in the hierarchy. The TPSN achieves two times better performance than RBS by time-stamping the radio messages in the Medium Access Control (MAC) layer of the radio stack and by relying on a two-way message exchange. The shortcoming of TPSN is that it does not estimate the clock drift of nodes, which limits its accuracy, and does not handle dynamic topology changes.

2.1.2 Uncertainties in radio message delivery

Non-deterministic delays in the radio message delivery in WSN can be magnitudes larger than the required precision of time synchronization. Therefore, these delays need to be carefully analyzed and compensated for. We shall use the following decomposition of the sources of the message delivery:

- Send time — time used to assemble the message and issue the send request to the MAC layer on the transmitter side. Depending on the system call overhead of the operating system and on the current processor load, the send time is nondeterministic and can be as high as hundreds of milliseconds. In our implementation this delay is deterministic, since we don't use MAC layer and have direct control over CPU and radio.
- Access time — delay incurred waiting for access to the transmit channel up to the point when transmission begins. The access time is the least deterministic part of the message delivery in WSN varying from milliseconds up to seconds depending on the current network traffic.
- Transmission time — the time it takes for the sender to transmit the message. This time is in the order of tens of milliseconds depending on the length of the message and the speed of the radio.
- Propagation time — the time it takes for the message to transmit from sender to receiver once it has left the sender. The propagation time is highly deterministic in WSN and it depends only on the distance between the two nodes. This time is

less than one microsecond (for ranges under 300 meters).

- Reception time — the time it takes for the receiver to receive the message. It is the same as the transmission time. The transmission and reception times overlap in WSN as pictured in Figure 2.1.
- Receive time — time to process the incoming message and to notify the receiver application. Its characteristics are similar to that of send time.

The RBS approach completely eliminates the send and access times, and with minimal OS modifications it is also possible to remove the receive time uncertainty. This leaves the mostly deterministic propagation and reception time in wireless networks as the sole source of error.

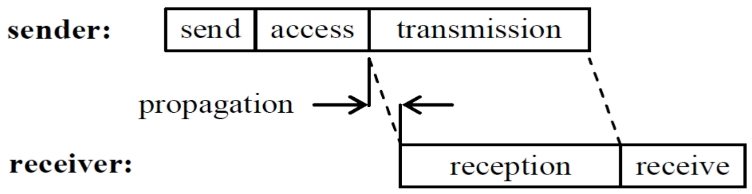

Figure 2.1. Decomposition of the message delivery delay over a wireless link.

With a two-way handshake of synchronization messages the TPSN protocol eliminates the unknown propagation time as well. Both the RBS and TPSN protocols suffer from the uncertainties of the overlapping transmission and reception times.

2.1.3 Flooding time synchronization protocol (FTSP)

The goal of the FTSP [3] is to achieve a network wide synchronization of the local clocks of the participating nodes. It is assumed that each node has a local clock exhibiting the typical timing errors of crystals and can communicate over an unreliable but error corrected wireless link to its neighbors. The FTSP synchronizes the time of a sender to possibly multiple receivers utilizing a single radio message time-stamped at both the sender and the receiver sides. MAC layer time-stamping can eliminate many of the errors. However, accurate time-synchronization at discrete points in time is a partial solution only. Compensation for the clock drift of the nodes is inevitable to achieve high precision in-between synchronization points and to keep the communication overhead low. Linear regression is used in FTSP to compensate for clock drift. Typical WSN operate in areas larger than the broadcast range of a single node; therefore, the FTSP provides multi-hop synchronization. The root of the network — a single, dynamically (re)elected node — maintains the global time and all other nodes synchronize their clocks to that of the root. The nodes form an ad-hoc structure to transfer the global time from the root to all the nodes, as opposed to a fixed spanning-tree based approach. This saves the initial phase of establishing the tree and is more robust against node and link failures and dynamic topology changes.

The FTSP utilizes a radio broadcast to synchronize the possibly multiple receivers to the time provided by the sender of the radio message. The broadcasted message contains the sender's time stamp which is the estimated global time at the transmission of a given byte. The receivers obtain the corresponding local time from their respective local clocks at message reception. Consequently, one broadcast message provides a synchronization point (a global-local time pair) to each of the receivers. The difference between the global and local time of a synchronization point estimates the clock offset of the receiver.

2.2 Dissemination protocols

Several dissemination protocols for WSNs exist. Their goal is to deliver data from a source node to the rest of the network within a short time period. They try to avoid simple broadcast retransmissions that lead to the broadcast storm problem. They also have to overcome obstacles that arise from the limited energy, computational power, and communication resources available to the sensors in the network.

2.2.1 Sensor protocols for information via negotiation (SPIN)

SPIN [5] is a family of adaptive protocols, that efficiently disseminates information among sensors in an energy-constrained wireless sensor network. Nodes running a SPIN communication protocol name their data using high-level data descriptors, called metadata. They use meta-data negotiations to eliminate the transmission of redundant data throughout the network. In addition, SPIN nodes can base their communication decisions both upon application-specific knowledge of the data and upon knowledge of the resources that are available to them. This allows the sensors to efficiently distribute data given a limited energy supply.

2.2.2 Trickle

Trickle [6] is an algorithm for propagating and maintaining code updates in wireless sensor networks. Trickle uses a “polite gossip” policy, where motes periodically broadcast a code summary to local neighbors but stay quiet if they have recently heard a summary identical to theirs. When a mote hears an older summary than its own, it broadcasts an update. Instead of flooding a network with packets, the algorithm controls the send rate so each mote hears a small trickle of packets, just enough to stay up to date. It is the most common dissemination algorithm for WSNs.

Such gossiping and meta-data negotiation approaches are however not well suited for the dissemination of small data, since they introduce overhead of similar size as the data. These dissemination protocols cannot be easily optimized for a fast periodic dissemination of only a few bytes within a short time window (e.g. due to metadata exchange).

2.2.3 Secondis

Secondis [1] differs from SPIN and Trickle dissemination protocols in several points. It is specifically designed to propagate short messages with a constant length, like synchronization messages. Secondis provides a probabilistic dissemination scheme that requires neither gossiping nor negotiation policies. It exploits the fact that the nodes get synchronized during the dissemination by using a modified slotted Aloha scheme to access the wireless medium, with the important addition that transmission probabilities adapt to changing network conditions.

Secondis separates synchronization activities from normal operation. All synchronization activities take place periodically in synchronization rounds. Normal operation takes place in frames between synchronization rounds. Frames are much longer, than synchronization rounds.

Figure 2.2 shows the state graph of a simplified discrete-time model that is executed by every node. Transitions between the states and variable updates are fully synchronized with the slots: they occur with zero delay during the time instant in which a slot ends and the next one begins. The time interval between the beginning of the transmission and the end of the reception of a message lies entirely within a single slot. A node starts its execution in the "wait" state, and remains in that state as long as no messages are received (signaled by $r \neq 1$) without transmitting any packet. Once a message is received, it keeps making transitions either to state "send" (with probability p) where it sends a packet, or to state "don't send" (with probability $1 - p$) where no transmissions are performed. The transmission probability of a node is given by $p = P_{init} \cdot (P_{df})^c$, where P_{init} is the transmission probability for the first message and P_{df} the decreasing factor for subsequent transmissions.

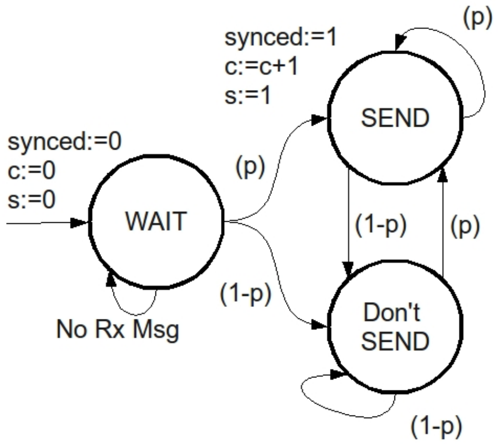

Figure 2.2. Discrete-time model of a node, for a single synchronization round.

Secondis follows the state graph depicted in Figure 2.2, except that Bernoulli trials are only performed every *k* - th time slot. This helps to reduce chances for collisions due to interferences from nodes out of the communication range (in general, the interference range is larger than the communication range). Simulation results highlight that sending every $k = 3$ slots yields the best results [1]. The synchronization round ends for all nodes after T_s time units. Additionally, a node is also allowed to finish its round earlier, i.e. as soon as the message has been forwarded C_{max} times, so that it can switch off the radio and save energy. Due to the decreasing transmission probability, chances for transmissions would get in any case very small after several transmissions. The root node has a special role. It sends its first message when its timer indicating the start of the synchronization round expires; it then keeps sending every *k* - th slot with probability one, until T_s time units have elapsed.

Chapter 3. Implementation

Below we present our software implementation of the Secondis protocol, featuring operating system for sensor modules – Contiki OS.

3.1 Contiki OS

The sensor devices are often severely resource constrained. An on-board battery or solar panel can only supply limited amounts of power. Moreover, the small physical size and low per-device cost limit the complexity of the system. Typical sensor devices are equipped with 8-bit microcontrollers, code memory on the order of 100 kilobytes, and less than 20 kilobytes of RAM. Contiki has been developed for such constrained environments. The operating system provides dynamic loading and unloading of individual programs and services. The kernel is event-driven, but the system supports cooperative multi-threading. Contiki is implemented in the C language and has been ported to a number of microcontroller architectures, including the Texas Instruments MSP430 [10]. This microcontroller is used on Tmote Sky sensor nodes.

Contiki makes use of protothreads [12], extremely lightweight, stackless type of threads that provide a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The advantage of protothreads over a purely event-driven approach is that protothreads provide a sequential code structure that allows for blocking functions, leading to cleaner code structure. The advantage of protothreads over ordinary threads is that a protothread does not require a separate stack. In memory-constrained systems, the overhead of allocating multiple stacks can consume large amounts of the available memory. Each protothread only requires between two and twelve bytes of state [12], depending on the architecture. A protothread runs within a single C function and cannot span over multiple functions.

3.1.1 Event scheduling

There are two types of timers in Contiki: event timers (from the *etimer* module) and real-time timers (from the *rtimer* module). The difference between the *etimer* and the *rtimer* is that an *etimer* posts an event to the process that set the timer on timer expiration. Therefore, the execution of timed actions is scheduled on timer expiration and it may take place several hundreds of milliseconds after the expiration, depending on the processor load. The exact execution time is not deterministic (as the load may vary). In contrast, a *rtimer* allows to execute timed actions in the order of a few microseconds. When a timer expires, the interrupt service routine calls immediately the callback associated to the timer, independently on what the processor was executing before the interrupt.

We use *rtimers* in our implementation, because our communication scheme requires high precision and delays in the order of milliseconds are not admissible.

3.2 Implementation of the Secondis state machine

As discussed in Section 2.2.3, in Secondis there are periodic synchronization phases. During each synchronization phase, time is subdivided into short time slots of constant length. Therefore, we decide to implement these two aspects by using two protothreads: *Sync* protothread and *Slot* protothread. This allows us to conveniently separate synchronization activities from synchronized activities into different protothreads. This decomposition is shown in the lower part of Figure 3.1.

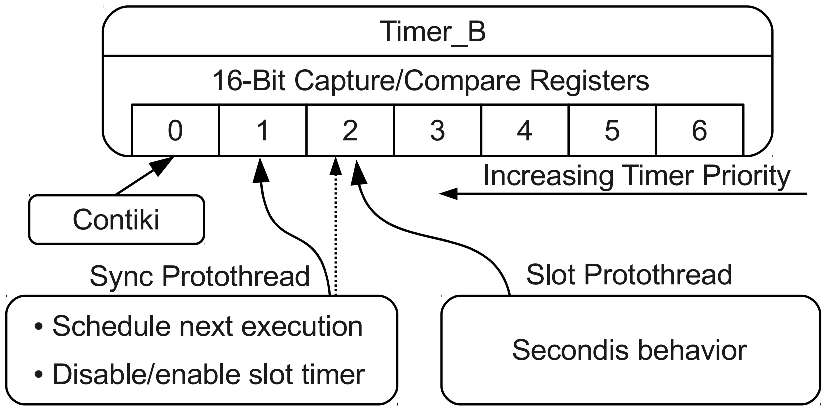
Timer_B
16-Bit Capture/Compare Registers
0
1
2
3
4
5
6
Increasing Timer Priority
Contiki
Sync Protothread
Slot Protothread
• Schedule next execution
• Disable/enable slot timer
Secondis behavior

Figure 3.1. Protothreads and timers mapping.

TMS430F1611 microcontroller, which is used on Tmote Sky wireless sensor modules, has two 16-bit timers Timer_A and Timer_B. The former has three capture/compare registers (CCRs) and the latter – seven CCRs. We use Timer_B in order to have 4 available CCRs for future use. In our implementation we make use of 2 CCRs: each protothread uses a specific CCR to schedule operations, as shown in the upper part of Figure 3.1. Modifications to Contiki code are made in order to support all available CCRs – we added functions that allow to init, set and disable a particular CCR in Timer_B. This allows us to simplify the code by making protothreads data independent, that is no global variables are required, since we don't need to share information between protothreads.

Parameter	**Explanation**	**Value**
T_F	Frame length (between two sync phases)	1 s
T_S	Sync phase length	200 ms
T_SLOT	Short time slot length	5 ms
P_{tx}	Transmission probability	0.56

Table 3.1. Secondis parameters.

Secondis is used to disseminate short time-stamped messages for time synchronization. The synchronization itself is done according to FTSP policy.

3.3 FTSP policy

Each synchronization message contains the *timeStamp* and the *seqNum*. The *timeStamp* contains the global time estimate of the transmitter when the message was broadcasted. The *seqNum* is a sequence number that is set and incremented by the root when a new synchronization phase is initiated.

Since all synchronized nodes periodically transmit synchronization messages, in a dense network a receiver may receive several messages from different nodes in a short time interval. Due to limited resources, an appropriate subset of the messages must be selected to create reference points. In our implementation an eight-element regression table stores the selected reference points used to calculate the drift of the local clock. A received synchronization message is used to create a reference point if the *seqNum* field is greater than the *seqNum* of previously received message. Thus, only the first message that arrived during a synchronization phase is used in the reference table. Old values in the table are replaced by new ones in circular order.

In our implementation, we have a fixed root node. The local time of the root node serves as the global time for the whole sensor network. The root is always synchronized and keeps transmitting time-stamped messages in each synchronization phase every third slot with probability P_{tx}. An ordinary network node gets synchronized when it receives at least four time-stamped messages. This minimum is required to perform linear regression for clock drift estimation. As soon as a node is synchronized, it starts acting according to the Secondis state machine (as shown in Figure 2.2).

On receiving an admissible time-stamped message, a node adds a reference point to its regression table and updates time conversion variables – *averageOffset* (offset between local and global time) and *skew* (clock drift estimation). The reference point is a triple *(messageSeqNum, messageLocalTime, messageTimeOffset)*. The last parameter is *messageGlobalTime – messageLocalTime*. Conversion from global to local time and vice versa is done according to (3.1).

$$\begin{gathered} globalTime = localTime + averageOffset \\ + skew * (localTime - localTimeAverage) \\ localTime = globalTime - averageOffset \\ - skew * (globalTime - averageOffset - localTimeAverage) \end{gathered} \tag{3.1}$$

3.4 Protothreads implementation

The Sync protothread (uses CCR 1, higher priority) is activated at the beginning and at the end of each synchronization phase. At the beginning of each synchronization phase, the Sync protothread enables CCR 2, that is used by the Slot protothread and schedules next execution of the Slot protothread. It also schedules its own next execution either in T_S or T_F milliseconds. The beginning of the next synchronization phase is scheduled using the global time, only if a node gets synchronized during current synchronization phase (according to FTSP policy, Section 3.3). We use global time in this case, because we schedule a long interval and a clock drift should be considered. Figure 3.2 shows flow-chart of the Sync protothread. The protothread turns radio module on and off in order to save power, when transmission is not required.

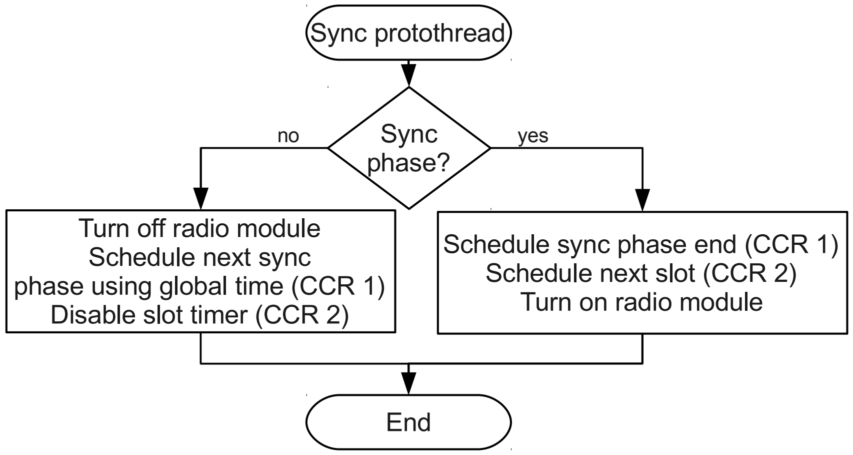
Sync protothread
Sync phase?
no
yes
Turn off radio module
Schedule next sync phase using global time (CCR 1)
Disable slot timer (CCR 2)
Schedule sync phase end (CCR 1)
Schedule next slot (CCR 2)
Turn on radio module
End

Figure 3.2. Sync protothread flow-chart.

The Slot protothread is executed at the beginning of each time slot during synchronization phase. This protothread uses CCR 2, that has lower priority, than CCR 1, used by the Sync protothread. Therefore, in case of simultaneous interrupts from both CCRs, the Sync protothread will be the first to execute. The state machine shown in Figure 2.2 is implemented within this protothread. Flow-chart of the protothread is shown in Figure 3.3. At the beginning of the execution a next slot is scheduled. Then we check if a node is synchronized (according to FTSP policy). A synchronized node can broadcast synchronization messages each third slot (see Section 2.2.3) with probability P_{tx}.

Floating point operations are very expensive in terms of time and therefore we can't use them in each slot. Instead, we implement all probability computations using integers. Contiki provides a function *random_rand()* that returns a 16-bit unsigned integer uniformly distributed from 0 to 65535. In order to transmit with certain probability, we generate random number with *random_rand()* and transmit time-stamped message only if generated integer is less than 65535 * P_{tx}.

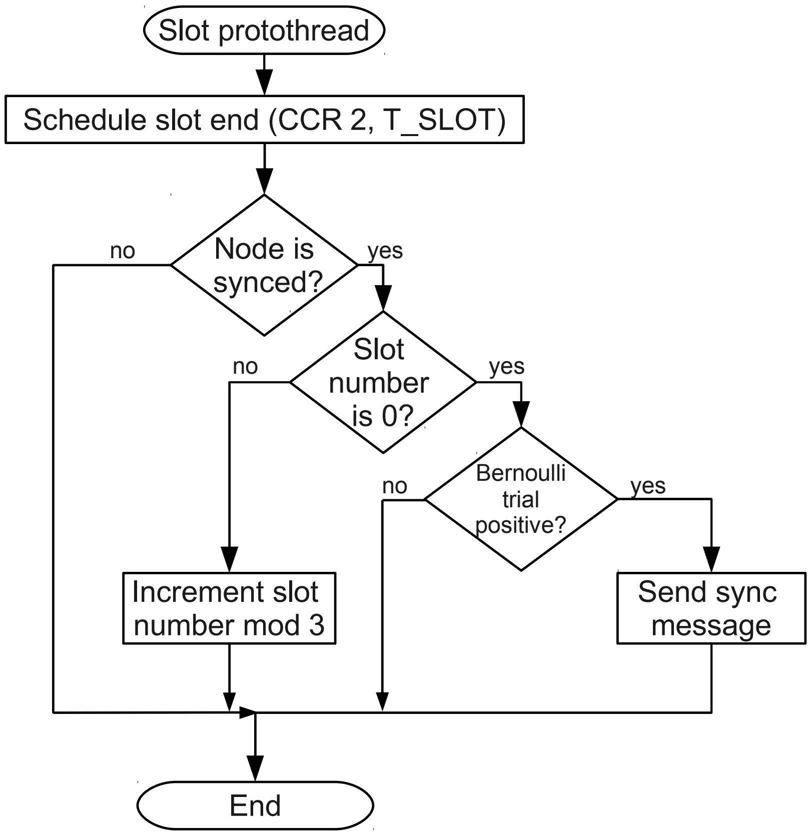
Slot protothread
Schedule slot end (CCR 2, T_SLOT)
Node is synced?
no
yes
Slot number is 0?
no
yes
Bernoulli trial positive?
no
yes
Increment slot number mod 3
Send sync message
End

Figure 3.3. Slot protothread flow-chart.

During a synchronization round a node receives first valid sync message and ignores all other message with the same sequence ID. According to FTSP policy, in order to be able to send synchronization messages further, a node needs to receive at least 4 different synchronization messages. Thus, at least first four synchronization phases only the root node broadcasts synchronization messages.

3.5 Receive procedure

Receive procedure is implemented within a callback function that is called from interrupt routine, serving radio interrupts. It is required to complete all computations as fast as possible in order to enable other interrupts. Since interrupt service routines can preempt protothreads, the receive procedure can be executed at any time during synchronization round (provided that there is incoming message). Flow-chart of receive procedure is shown in Figure 3.4.

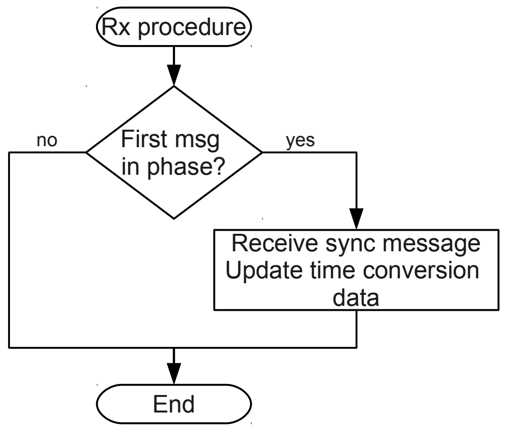

Figure 3.4. Receive procedure flow-chart.

A message is accepted for further processing if this is the first time-stamped message in current synchronization phase. Received time information is added to FTSP regression table and conversion parameters are updated, if there are enough reference points in the table. If a node is synchronized, then starting from next slot it tries to broadcast time-stamped messages, according to state machine in Figure 2.2.

Figure 3.5 summarizes activities, that take place during different rounds.

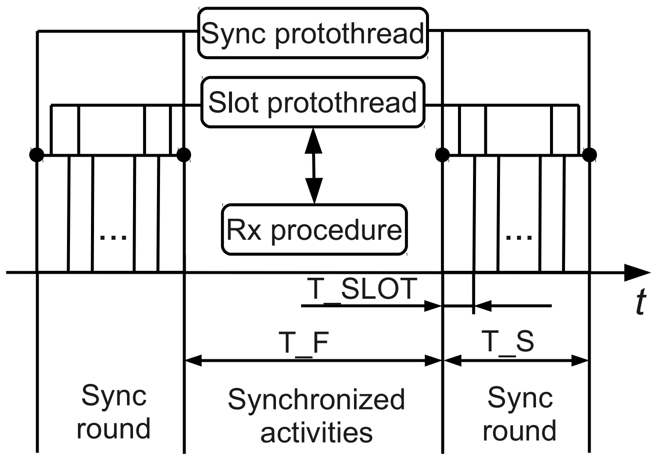

Figure 3.5. Sequence of protothreads′ execution.

The Sync protothread is executed at the beginning and at the end of each synchronization phase. The Slot protothread is executed in each time slot within synchronization phase. The Slot protothread may be interrupted by receive procedure. A synced node performs Bernoulli trials and decides weather to broadcast time-stamped messages each third time-slot after reception of a time-stamped message.

Chapter 4. Evaluation

4.1 Setup

We evaluate our software on the PC using COOJA simulator [8] that comes with Contiki OS. We run simulations for two different network topologies – a line with six nodes and a two-hop star (four nodes in each hop). We run experiments for different transmission probabilities and count the number of missed synchronization rounds.

4.1.1 COOJA simulator

COOJA is a flexible Java-based simulator initially designed for simulating networks of sensors running the Contiki operating system. COOJA simulates networks of sensor nodes where each node can be of a different type; differing not only in on-board software, but also in the simulated hardware. COOJA is flexible in that many parts of the simulator can be easily replaced or extended with additional functionality. A simulated node in COOJA has three basic properties: its data memory, the node type, and its hardware peripherals. The node type may be shared between several nodes and determines properties common to all these nodes. For example, nodes of the same type run the same program code on the same simulated hardware peripherals. Nodes of the same type are initialized with the same data memory, except for the node ID.

COOJA allows to vary such communication parameters, as transmission range, interference range, transmit and receive success probability. In our simulations we use double interference range, compared to transmit range, and 100% transmit and receive probability.

4.1.2 Tmote Sky platform

Tmote Sky is an ultra low power wireless module for use in sensor networks, monitoring applications, and rapid application prototyping. By using industry standards, integrating humidity, temperature, and light sensors, and providing flexible interconnection with peripherals, Tmote Sky enables a wide range of mesh network applications. Tmote Sky features high performance, functionality, and expansion. With Contiki OS support out-of-the-box, Tmote leverages emerging wireless protocols. Figure

4.1 shows the module.

The low power operation of the Tmote Sky module is due to the ultra low power Texas Instruments MSP430F1611 microcontroller featuring 10 kB of RAM, 48 kB of flash, and 128 B of information storage. This 16-bit RISC processor features extremely low active and sleep current consumption that permits Tmote to run for years on a single pair of AA batteries. The MSP430 has 8 external ADC ports and 8 internal ADC ports. The ADC internal ports may be used to read the internal thermistor or monitor the battery voltage. A variety of peripherals are available including SPI, UART, digital I/O ports, watchdog timer, and timers with capture and compare functionality (Timer_A with 3 CCRs and Timer_B with 7 CCRs).

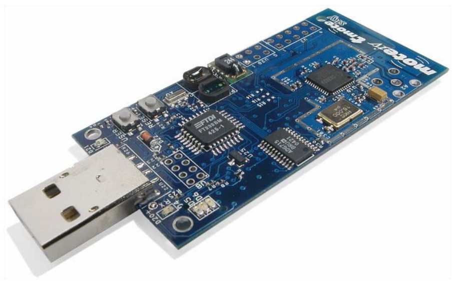

Figure 4.1. Tmote Sky module.

4.2 Results

4.2.1 Line topology

We first run a series of experiments for a line topology of six nodes. Figure 4.2 shows COOJA simulation window for this topology. Transmission and interference ranges are shown for one of the nodes. The inner circle corresponds to transmission range, and the outer – to the interference range.

We use three different constant transmission probabilities P1 = 0.3, P2 = 0.56, P3 = 0.9 in our experiments. We are interested in number of missed synchronization rounds under each transmission probability. Table 4.1 shows measured amount of missed synchronization rounds (in %) during simulation time of 1000000 ms, corresponding to around 640 synchronization rounds. Figure 4.3 shows amount of missed synchronization rounds (in %) for all nodes under transmission probability P2.

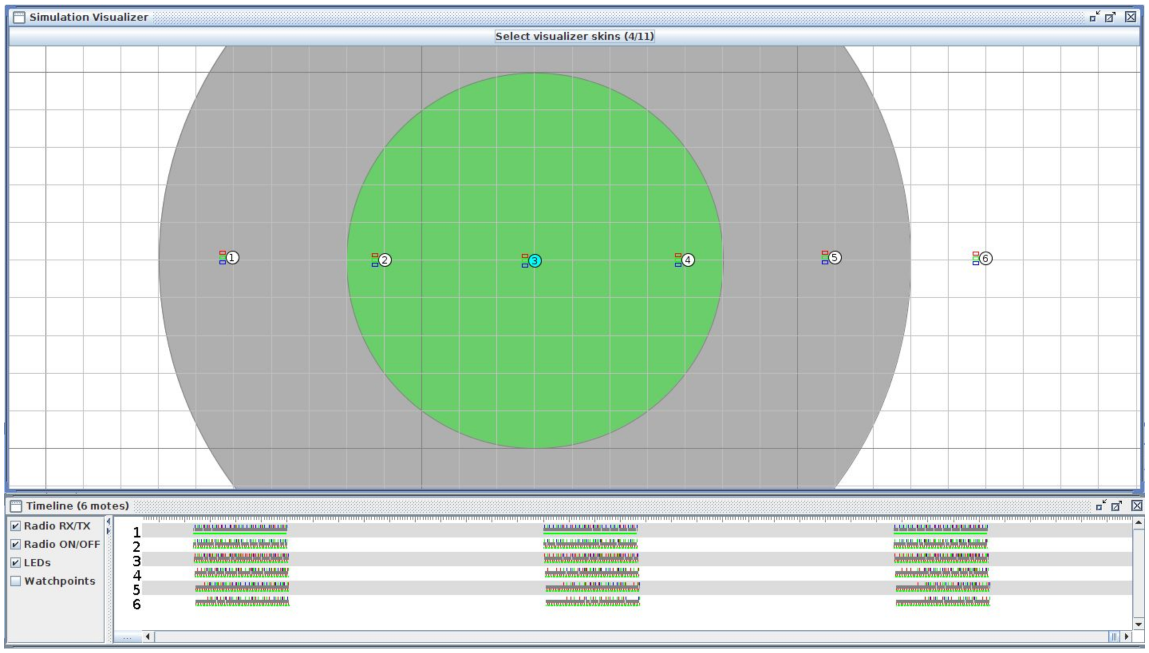

Figure 4.2. Simulation for line topology, transmission and interference ranges.

Lower part of the Figure 4.2 shows that synchronization rounds are periodically and synchronously executed on all nodes.

	Missed rounds, %		
Node	P1 = 0.3	P2 = 0.56	P3 = 0.9
2	1.88	1.88	1.88
3	5.94	5.16	5.78
4	9.22	7.81	9.53
5	13.44	11.25	12.81
6	20.94	15	16.09

Table 4.1. Missed rounds under different transmission probabilities (line).

Obviously, number of missed rounds increases with number of hops. Since interference range is twice as large, as transmission range, we get worse results for P3, than for P2, because of increased number of interferences. We see that transmission probability P2 is near optimal (P2 < P1 and P2 < P3). That confirms results derived in [1], according to which, transmission probability equal to 0.56 is optimal, in case of constant transmission probability.

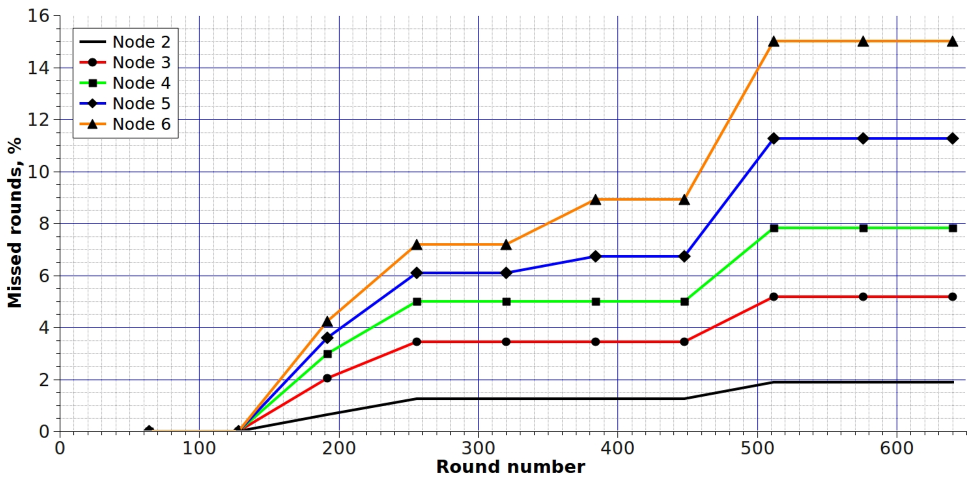

Figure 4.3. Missed rounds (%) under P2 (line).

4.2.2 Star topology

We use the same communication settings and the same set of probabilities to run experiments on the star topology. We use a two-hop star, that has four nodes in each hop, nine nodes total (with the root). Figure 4.4 shows COOJA simulation window for the star topology, transmission and interference ranges are shown for the root node.

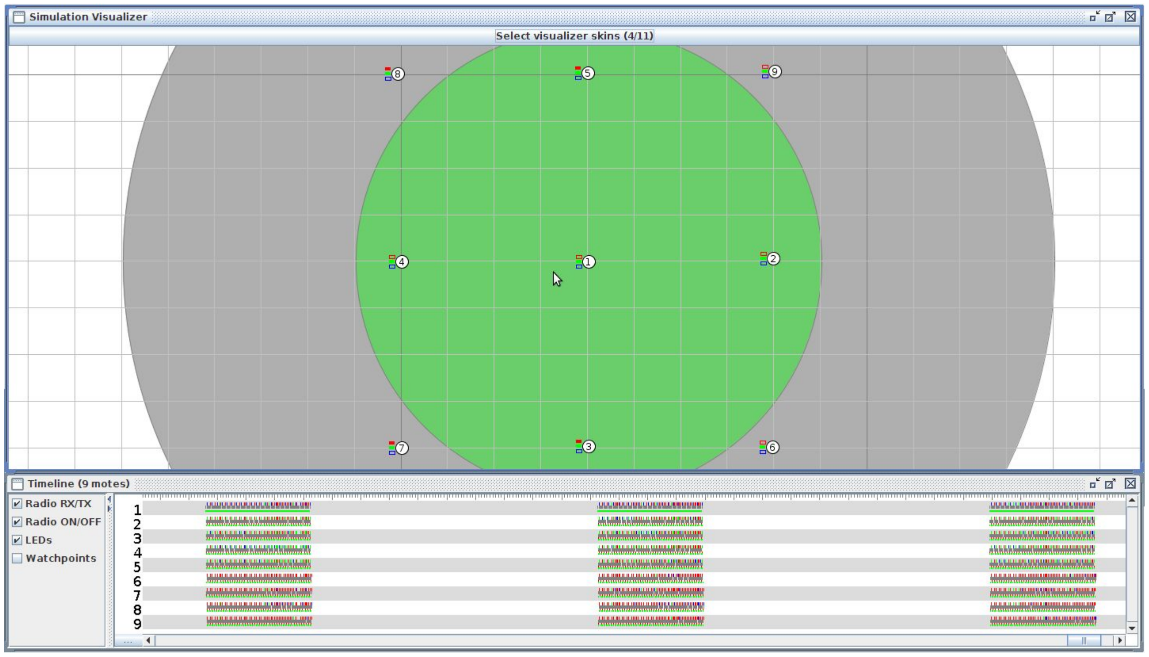

Figure 4.4. Simulation for star topology, transmission and interference ranges of the root node.

Table 4.2 shows number of missed synchronization rounds (in %) for three different transmission probabilities. Nodes 2, 3, 4 and 5 are in one hop distance from the root and the rest of the nodes are in two hop distance from the root. In this case number of missed rounds depends on the distance from the root. All nodes that are equidistant from the root have approximately the same amount of missed rounds. The number of missed rounds increases dramatically with the number of hops (due to interferences from other nodes). Figure 4.5 shows missed rounds for transmission probability P2.

	Missed rounds, %		
Node	P1 = 0.3	P2 = 0.56	P3 = 0.9
2	1.88	1.88	1.88
3	1.88	1.88	1.88
4	1.88	1.88	1.88
5	1.88	5.63	1.88
6	7.5	13.44	59.84
7	6.56	13.75	47.97
8	5.47	12.81	53.28
9	6.41	13.13	53.28

Table 4.2. Missed rounds under different transmission probabilities (star).

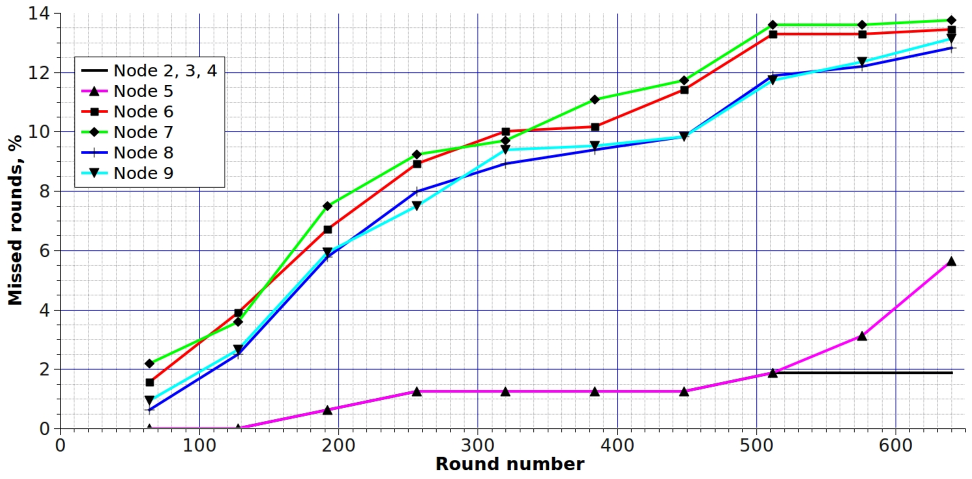

Figure 4.5. Missed rounds under P2 (star).

Obviously, the amount of missed rounds is proportional to the distance to the root node (number of hops). The amount of missed rounds linearly increases with the transmission probability (Table 4.2). Judging by the obtained results, we can conclude that optimal transmission probability for this topology lies below 0.3. But low transmission probability means long synchronization. In order to cope with high interference level and shorten synchronization time, it might be useful to introduce adaptive transmission probability.

4.2.3 Summary

We conducted experiments for two topologies using three constant transmission probabilities. Obtained results show, that successful dissemination depends not only on the transmission probability but on the network topology and communication parameters. Each topology has its optimal transmission probability, mainly because of different interference levels. Figure 4.6 shows average number of missed synchronization packets (in %) for both topologies and different transmission probabilities.

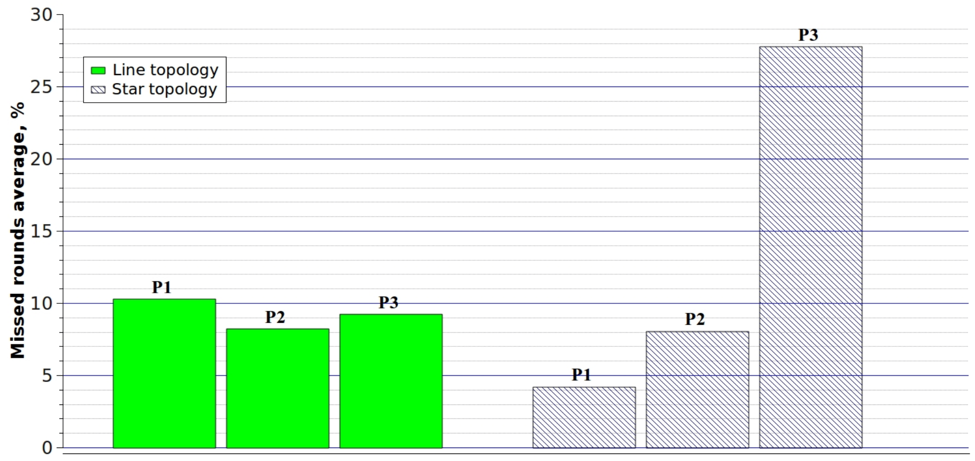

Figure 4.6. Average number of missed rounds (in %) for both topologies.

In order to achieve both fast and reliable dissemination and low interference level, we could use adaptive transmission probabilities (decrease transmission probability with each sent time-stamped message within one synchronization round) and adaptive node importance levels (nodes that are not important do not transmit time-stamped messages).

Chapter 5. Conclusion

We implemented the core features of the Secondis dissemination protocol in the Contiki OS. We exploit Contiki's multitasking features to implement the protocol's behavior in a clean and efficient way. We use the COOJA simulator to evaluate our implementation based on the Tmote Sky wireless sensor module as the target platform. Our simulations on different network topologies demonstrate that in order to achieve fast and reliable dissemination in any network, a protocol has to be adaptive, since different networks have different optimal transmission probabilities.

Bibliography

[1] F. Ferrari et al., “Secondis: an adaptive dissemination protocol for synchronizing wireless sensor networks”

[2] S. Ganeriwal et al., “Timing-sync protocol for sensor networks,” in SenSys’03, New York, USA, 2003.

[3] M. Maroti et al., “The flooding time synchronization protocol,” in SenSys’04, New York, USA, 2004.

[4] C. S. Raghavendra, K. M. Sivalingam, T. Znati, Wireless Sensor Networks. Springer Science+Business Media, New York, 2006, ch. Dissemination protocols for large sensor networks.

[5] W. R. Heinzelman, J. Kulik, and H. Balakrishnan, “Adaptive protocols for information dissemination in wireless sensor networks,” in MobiCom’99, New York, USA, 1999.

[6] P. Levis et al., “Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks,” in NSDI’04, Berkeley, USA, 2004.

[7] A. Dunkels et al., “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”.

[8] J. Eriksson et al., “COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks”.

[9] J. Polastre et al., “Telos: Enabling Ultra-Low Power Wireless Research”.

[10] Texas Instruments, MSP430x1xx Family User's Guide, www.ti.com/msp430.

[11] Moteiv, Tmote Sky Low Power Wireless Sensor Module Datasheet, www.moteiv.com.

[12] A. Dunkels et al., Contiki 2.x Reference Manual, www.sics.se.