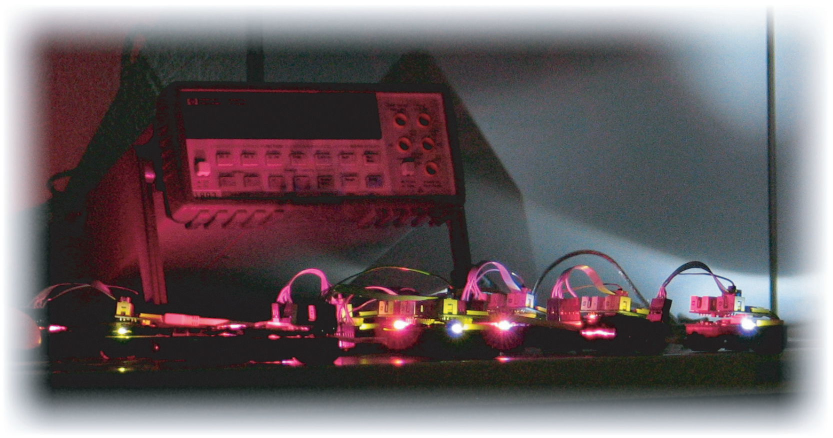
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Wireless Fire Sensor Network Demonstrator

Roman Lim



MASTER'S THESIS
MA-2006-14

Summer term 2006

Supervisors: Andreas Meier
Dr. Jan Beutel
Professor: Dr. Lothar Thiele

Start Date: 21st April 2006
Issue Date: 20th October 2006

Abstract

Wireless sensor network applications face the problem of exchanging messages over an unreliable network. For safety important applications like wireless fire sensor networks, this characteristic assigns a hard task.

This thesis implements a highly reliable monitoring system that uses low power, addressing the fact, that nodes in a wireless sensor network are mostly battery powered. We achieved reliability with redundant information flow and acknowledged messaging. Nodes were monitored with a round based heartbeat scheme and media access was implemented in a central controlled TDMA scheme.

The implementation was done on a state of art wireless sensor platform, the Tmote Sky. This platform features a 2.4GHz transceiver and a low power microcontroller. The implementation was done based on TinyOS 2.0, a framework for wireless sensor network applications.

The Deployment Support Network(DSN), developed at ETH, provided helpful support for the implementation process of the application. In order to use it, we built a BTnode-Tmote Sky DSN-Adapter. This Adapter consists of a hardware connector, a software logging component for TinyOS 2.0 and a bootstrap loader for the Tmote Sky.

With help of the DSN, we profiled the implemented monitoring system. The calculated lifetime estimation amounts 0.96 to 2.65 years.

The developed DSN-adapter for Tmote Sky enables DSN support for further work on this platform.

Contents

1: Introduction *1*

1.1 Motivation 1

2: Related work *3*

2.1 Low Power 3

2.2 Network Supervision 4

3: Prototype Platform and Operating System *5*

3.1 Tmote Sky 5

3.1.1 Microcontroller MSP430 6

3.1.2 Transceiver Chipcon 2420 6

3.2 TinyOS 2.0 7

3.2.1 Components Model 7

3.2.2 Concurrency 8

3.2.3 Low Power Support for Microcontrollers 8

3.2.4 Resource Arbitration 9

3.2.5 TinyOS MAC Layer Implementation on Tmote Sky 10

3.3 Measurements 10

3.3.1 Time and Current 10

3.3.2 Radio Transmission 11

3.3.3 Lifetime Estimation Model 15

4: Connection to the Deployment Support Network *17*

4.1 Introduction 17

4.2 Deployment Support 17

4.2.1 Communication over the Target Network 17

4.2.2 Communication over a Wired Overlay Network 18

4.2.3 Communication over a Wireless Overlay Network 18

4.3 Target Connection for the Tmote Sky 19

4.4 Target Programming 19

4.4.1 Challenges 19

4.4.2 Realisation of the Bootstrap Loader 21

4.4.3 Installation of the Bootloader 23
4.5 Communication 23
4.5.1 Challenges 23
4.5.2 DSN Component 25
4.6 Tmote Sky Target Adapter Hardware 27

5: Implementation of Status Monitoring *29*
5.1 Failure Detection Strategies 29
5.2 Approach 30
5.2.1 Requirements and Assumptions 30
5.3 Concepts 31
5.3.1 Robustness 31
5.3.2 Meeting Time Constraints 32
5.3.3 Low Power 34
5.4 Implementation Details 34
5.4.1 Component Overview 34
5.4.2 Time Synchronisation 34
5.4.3 Radio Configuration, CC2420 Stack Changes 37
5.4.4 TDMA Slot Duration 37
5.4.5 Start-up Phase 41
5.5 Tests and Analysis 41
5.5.1 Testbed 41
5.5.2 TDMA Schedule Verification 42
5.5.3 Reporting Time Measurement 43
5.5.4 Power Consumption Estimation 43

6: Conclusion and Outlook *49*
6.1 Contributions 49
6.2 Further Work 50

A: Aufgabenstellung *51*

Tables

3-2 Overview of the available low power modes of the MSP430 microcontroller 9
3-3 Current consumption of the Tmote Sky in different low power modes, CC2420 radio module is switched off 11
3-5 Parameters of the power estimating model 15
3-6 Values of the model. Each is an average of measured values 16

4-2 Comparison of different programming methods for the MSP430 20
4-3 Log levels of JAWS 26
4-4 Symbols for numerical logging 27
4-5 Tmote 6-pin Expansion (U28) 27
4-6 Tmote 10-pin Expansion (U2) 28

5-1 Values for the calculation of slot duration 38
5-2 Measured application processing time 40
5-3 Parameters for the round count experiment. 45
5-4 Average currents for n rounds per period 46
5-5 Calculated network lifetime with a 2300mAh battery. Theoretical bounds and realistic values, based on round measurements. 47

Figures

3-1 MoteIV's Tmote Sky 6
3-2 Example of the wiring in a TinyOS program 8
3-3 IEEE 802.15.4 Packet format 10
3-4 Setup of the time measurement experiment 12
3-5 Time for sending a packet, including radio startup time 12
3-6 Current curve of the Tmote Sky node when transmitting a packet . . . 13
3-7 Current curve of the Tmote Sky node when receiving a packet 14

4-1 DSN with targets 18
4-2 Memory organisation of the MSP430F1611 21
4-3 Bootstrap loader entry sequence 22
4-4 Flowchart of the bootstraploader for Tmote Sky. 24
4-5 Available connectors to the Tmote Sky 25
4-6 Fotos of the Tmote Sky DSN-Adapter 28

5-1 Overview on the recovery mechanism 31
5-2 TDMA scheme for one wave round 33
5-3 Wiring of theimplementaton of the failure detector 35
5-4 Timestamping with the CC2420 on a packet while sending 36
5-5 Offset deviation with MAC-layer timestamping and unidirectional clock synchronisation 37
5-6 Transmission time with worst case clock drift 39
5-7 Measured application layer processing time of the acknowledge packets 39
5-8 Arrangement of noded in the testbed 41
5-9 Output of the packet sniffer in a network of 5 nodes 42
5-10 Histogram of the measured reporting times with different periods T_{Period} 43
5-11 Plot of the round count experiment at a transmission power parameter of 1 44
5-12 Plot of the round count experiment at a transmission power parameter of 2 44
5-13 Plot of the round count experiment at a transmission power parameter of 3 45

1

Introduction

1.1 Motivation

Since a few years, wireless sensor networks are an emerging research field. The ability to deploy a huge amount of cheap nodes almost everywhere and to gather sensor data for evaluation creates new possibilities for applications. Already established applications can be re-implemented based on approaches using such a wireless network in order to decrease the need for infrastructure (e.g. wires).

An interesting application is a wireless fire sensor multi-hop network. There is an on-going industrial project that aims to build a reliable fire sensor system based on a wireless multi-hop network.

The goal of this thesis is, to implement a demonstrator that addresses the challenges of a reliable reporting system as it is needed in a wireless fire sensor network: Time constraints, robustness and low power consumption.

While developing and implementing the sensor application, this work is also intended to gain practical experiences with the JAWS deployment support network. This supporting tool offers help developing and deploying wireless sensor networks.

2 Related work

The area of wireless sensor networks (WSN) has become a very popular research area in the last few years. A system built of a lot of independent, low cost computing units, interacting wirelessly with each other and gathering sensor data can be applied for a huge number of applications. One important advantage of such networks is the little need for infrastructure. Sensors can be deployed virtually everywhere with extremely little effort compared to wired sensor systems. Recent sensor projects, as example, were deployed in places like glaciers [36, 37], islands [39] or on a volcano [38].

2.1 Low Power

Sensor nodes are usually powered by normal batteries, so low power consumption is a main requirement for applications, where a long network lifetime is needed. This requirement had been addressed by several MAC layer protocols for wireless sensor networks. The main strategies for saving power are[1]:

Avoid collisions Packet collision leads to information loss. The packet has to be retransmitted, which uses additional energy to send.

Avoid overhearing In WSNs, transmissions of a sender reaches all its neighbouring nodes. Nodes that hear a packet do normally not know the intended receiver a priori. They all have to start receiving each packet. If the message is for another node, the used energy for receiving data is unnecessarily wasted.

Minimise idle listening Nodes that do not know, when a packet is expected to arrive, spend most of the time on idle listening.

Minimise protocol overhead For co-ordinated medium access, there is additional information sent over the network. These transmissions do not bring information for the upper layer application but need energy. A lot of WSN applications do only send a few bytes per packet, so the overhead added by the MAC protocol can be a big part of the transmitted data.

MAC layer protocols for WSN applications can be roughly divided into contention based [2, 3, 4] and schedule based protocols [5, 6, 7, 8, 9, 10]. The contention based approach mostly has only a small protocol overhead, but does not completely eliminate the first three points of the strategies described previously. TDMA protocols are inherently collision free because every transmission is scheduled, but the protocol overhead increases.

2.2 Network Supervision

For applications like fire sensing or intrusion detection, it is essentially to ensure a fully serviceable system, or at least to get alarmed when a node is not available anymore. For such scenarios, node supervision is an important task.

An approach to detect failures in distributed systems is to use a network heartbeat. Every node periodically announces its heartbeat, which is then diffused across the network. A node is considered to have failed, if the last known heartbeat excesses a specified age. Wang et al.[11] analysed heartbeat failure detectors that distribute the heartbeat in a gossip style in wireless ad-hoc networks. Their aims were to provide distributed, effective and robust mechanisms for failure detection. Two gossiping schemes were simulated:

Linear scheme The order of nodes that send their heartbeat is fixed by the identifier of each node. Gossiping is done in rounds. At a network diameter of D, gossiping will be completed (the information is diffused to every network member) in at most D rounds.

Two-phase scheme This scheme divides gossiping in two phases. An inward phase and an outward phase. Each edge between two nodes gets a direction. Nodes with no inward links send their messages first. As soon as a node has received messages from all inward pointing edges, it sends its own message. When all inward messages are sent, the second phase begins and all edge directions are reversed.

Their simulation showed, that in static networks, the advantage of the two-phase over the linear scheme gets clear, when it comes to sparser networks: The heartbeat information gets distributed faster.

Tai et al.[12] use clusters to organise the distribution of heartbeat information. They make use of the fact, that transmissions can be overheard in WSNs from neighbouring nodes to increase robustness when gathering alive information. Like in [11], the heartbeat scheme is done in two phases. Different in this approach is to divide the network into clusters. All nodes in a cluster can communicate directly with the cluster head. Detected failures are then forwarded between clusters.

In this thesis, different to the previously presented work, we implemented such a reporting system on a real sensor network based on Tmote Sky. Special attention was paid on low power consumption.

3

Prototype Platform and Operating System

This chapter gives an overview on the used hardware platform. Besides general information, we give more detailed insight into the mechanisms of the radio communication and how low power concerns are addressed. These two issues are important to understand the details of the later implementation. In a second part, characteristic behaviour and power consumption are measured.

For software development we used TinyOS 2.0 [13], a popular framework for wireless sensor network applications. We explicate the mechanisms that make TinyOS suitable for this task.

3.1 Tmote Sky

The Tmote Sky is a state of art sensor node for wireless sensor networks. This platform, formerly named Telos [15], was developed at UC Berkeley. Since 2005 it is commercially distributed by the Moteiv Corporation [14]. The key features of the Tmote Sky platform are [16]:

- 250kbps 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver
- Interoperability with other IEEE 802.15.4 devices
- 8MHz Texas Instruments MSP430 microcontroller (10k RAM, 48k Flash)
- Integrated ADC, DAC, Supply Voltage Supervisor, and DMA Controller
- Integrated onboard antenna with 50m range indoors / 125m range outdoors
- Optional Integrated Humidity, Temperature, and Light sensors
- Ultra low current consumption
- Fast wakeup from sleep ($< 6\mu s$)
- Hardware link-layer encryption and authentication

- Programming and data collection via USB
- 16-pin expansion support and optional SMA antenna connector
- TinyOS support : mesh networking and communication implementation
- FCC modular certification : conforms to all US and Canada regulations

For this thesis the most interesting features were the power consumption and the characteristics of the radio module.

3.1.1 Microcontroller MSP430

The microcotroller [17] of the Tmote Sky module features extremely low active and sleep current consumption. In the active mode, the microcontroller is driven by the internal digitally controlled oscillator (DCO) that may operate up to 8MHz. The DCO may be turned on from sleep mode in 6s. When the DCO is off, the MSP430 operates of an external 32.768kHz watch crystal. In the later implementation, we will use this crystal clock source to maintain synchronisation between nodes.

3.1.2 Transceiver Chipcon 2420

The radio module on Tmote Sky is a single chip 2.4GHz IEEE802.15.4 compliant RF transceiver [18]. It provides a data rate of 250 kbps and hardware support for packet handling, data buffering, clear channel assessment and link quality indication. If a packet conforms to the IEEE standard [19], address recognition can be done in hardware. A packet is accepted, if the address fields (address of receiver, personal area network indentifier) of the packet correspond to the configuration of

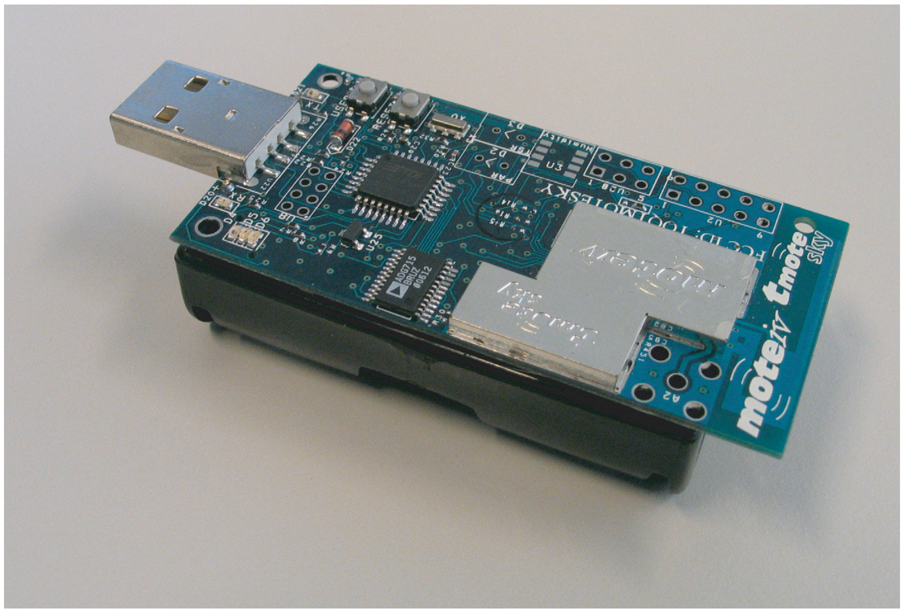


Figure 3-1
MoteIV's Tmote Sky

the radio module. Otherwise the received data is ignored. This feature avoids unnecessary computing steps in the attached microcontroller for overheard packets. The transmitted packets do not have to comply with the IEEE standard. Sacrificing some hardware support for packet handling, most of the packet frame can be freely defined. Mandatory is only the length field of the packet. For receiver synchronisation, a preamble of 0 to 16 bytes can be specified, followed by the start of frame delimiter (SFD). The reception of a SFD triggers a pin on the CC2420 which can be used to capture the arrival time of a packet. The integrated link quality indication is calculated based on the correlation of the 8 first symbols received of a packet. This value represents a sort of chip error rate. The payload of one packet can get as big as 127 bytes.

A special feature of the packet based CC2420 supports timestamping of sent packets. The transceiver allows to modify a packet in the radio FIFO even when the process of transmitting already has started. Therefore, a packet can be timestamped with the measured time at start of the transmission.

3.2 TinyOS 2.0

Software for embedded systems, especially for wireless sensor networks, has to satisfy needs that are very different from those of personal computers. The most important issues are the limited system resources of a node, namely energy supply, computing power and memory. TinyOS [13] is a standard framework for developing WSN applications. The final release of the 2.0 version is planned for November 2006. This new version has valuable improvements [20] compared to TinyOS 1.x. Overall stability has been increased at the expense of some flexibility. Some important changes concerned the scheduler and the stability of the network stack. The modular composition of TinyOS allows to build reusable components and programs that have a very small footprint. Applications are written in a special C language called NesC. This programming language contains support for the concurrency model of TinyOS and also the component based application architecture. In the following sections, a description of the basic concepts of TinyOS is given.

3.2.1 Components Model

A TinyOS program consists of several modules, wired together by a configuration. Figure 3-2 shows an example for this modular composition. A single module or configuration is called a component. Such components implements defined interfaces for interaction with other components. This leads to easily replaceable parts of code, without having to make changes to the whole program. The advantages of this model are also applied when it comes to portability to other node platforms. There are several levels of hardware abstraction foreseen in the concept of TinyOS [21], each level modelled as an component. The hardware presentation layer (HPL) represents the lowest level interface to the hardware. At this level, the functionality of the interface depends completely on the capabilities of the hardware. No states are stored and also no resource arbitration is done. At the next level, the hardware adaptation layer (HAL), useful abstractions are built on top of the HPL. The strategy is to give an interface which does not compromise efficiency for convenience. Software which

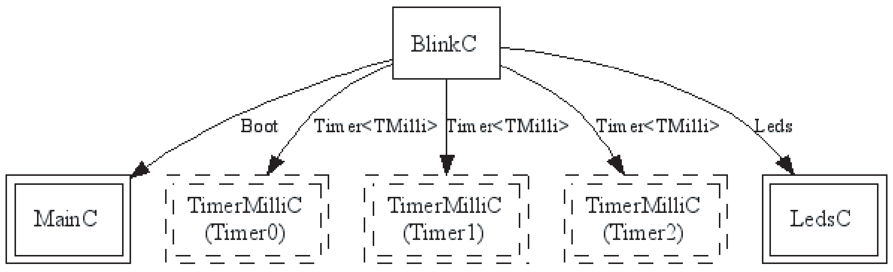


Figure 3-2
Example of the wiring in a TinyOS program [13]. The BlinkC program lets the LEDs flash according to a counter. It is the NesC equivalent to the famous 'Hello Word' program.

implements time or performance critical actions can wire to this layer. This layer saves its state and has the functionality to prevent simultaneous accesses from disturbing each other. The highest grade of abstraction is provided by the hardware independent layer (HIL). Obviously, this telescoping abstraction makes it very comfortable to write software for new hardware components, e.g. different microcontrollers or flash memory chips. Components, which do only use the API of the HIL of a specific hardware module do not need to be rewritten.

3.2.2 Concurrency

The transformation of concurrent processes to software is an important task of an operating system. In wireless sensor networks this concerns mostly communication on several devices (radio, UART, ..) and events coming from sensors or timers. On systems with less limitations, the processing of such concurrent events is realised with threads. This solution results for every thread in an overhead for saving states and other information in the stack. Each context switch needs computation power for saving register contents. The fact that programs on sensor nodes are quite small, the designers of TinyOS 2.0 implemented concurrency in a much simpler way. There are no threads. Each time a task is started, it runs to completion. This behaviour is referenced to as synchronous execution. Only hardware interrupts can pre-empt a running process. Because no context switches are made when running such interrupt code (asynchronous), the programmer has to take precaution for shared variables. NesC allows to declare a section of code to be atomic. Those sections can not be interrupted. This approach makes it necessary to keep tasks and atomic sections short in order to have fast reaction times for interrupts. The NesC preprocessor includes a race detection system which throws warnings when variables are not well protected.

3.2.3 Low Power Support for Microcontrollers

Most microcontrollers support low power applications with several sleep modes. Tmote's microcontroller has five low power states, as shown in table 3-2. Low power mode LPM0 for example disables only the CPU and the main system clock. In LPM4 the microcontroller can only be woken up by an external interrupt because all timers are off.

Every time the task queue is empty, the scheduler of TinyOS puts the microcontroller in the deepest possible low power mode. This mode is determined by a chip-specific low power state calculation function. So no explicit command is needed in TinyOS to let the MSP430 sleep. A detailed description of this feature is available at the TEP 112 [23]. Other hardware than the microcontroller have to be stopped explicitly to save energy.

Mode	**CPU and Clocks Status**
Active	CPU is active, all enabled clocks are active
LPM0	CPU, MCLK are disabled SMCLK , ACLK are active
LPM1	CPU, MCLK, DCO osc. are disabled DC generator is disabled if the DCO is not used for MCLK or SMCLK in active mode SMCLK , ACLK are active
LPM2	CPU, MCLK, SMCLK, DCO osc. are disabled DC generator remains enabled ACLK is active
LPM3	CPU, MCLK, SMCLK, DCO osc. are disabled DC generator disabled ACLK is active
LPM4	CPU and all clocks disabled

Table 3-2: Overview of the available low power modes of the MSP430 microcontroller [17]

3.2.4 Resource Arbitration

Resources, for instance timers, busses or ADCs, can be used by several components in a TinyOS program. On way to deal with this is to create a virtual abstraction of the resource in software. This enables a multiplexing of the underlying resource. However, resources like busses need to grant full access to a bus-client. Such a shared resource needs arbitration. For this purpose, TinyOS provides an arbiter component [22] that can be wired into a program when needed. Every client that wants to access the resource has to request it first from the arbiter. If the resource is free, the arbiter signals a granted event to the requestor. The resource has to be released by the client, before another one can use it.

This concept is used on the Tmote Sky for arbitrating the access to the shared bus of the flash chip (storage) and the radio (network). In each case, the storage and network stacks need exclusive access to the bus when using it, but they also need to share it with the other subsystem.

There is no support for deadlock prevention. The programmer has to take care of it. In order not to block other processes, a resource should be acquired as short as possible. Additionally to the arbitration, the arbiter knows always, whether the resource is in use. This can be used to resources into power save modes whenever they are not needed.

3.2.5 TinyOS MAC Layer Implementation on Tmote Sky

TinyOS 2.0 has implemented a contention based MAC protocol. No low power considerations are made. Whenever a packet is to be sent, the channel is first sensed by utilising the clear channel assessment (CCA) functionality of the CC2420 chip. If the received signal power is over a programmable threshold, it is assumed, that the channel is in use and a random congestion backoff timer is set in order to sense the channel at another point in time. If the channel is clear, the packet is sent. No collision avoidance is implemented but there exist an acknowledgement mechanism. If acknowledges are enabled, the network stack keeps track on the acknowledged packets. On the receiving end, an acknowledge packet is automatically sent by the Chipcon radio module if the corresponding flag in the MAC header of the packet is set. The link-layer implementation of TinyOS uses a IEEE 802.15.4 compliant MAC header in order to take advantage of the hardware address recognition of the radio chip.

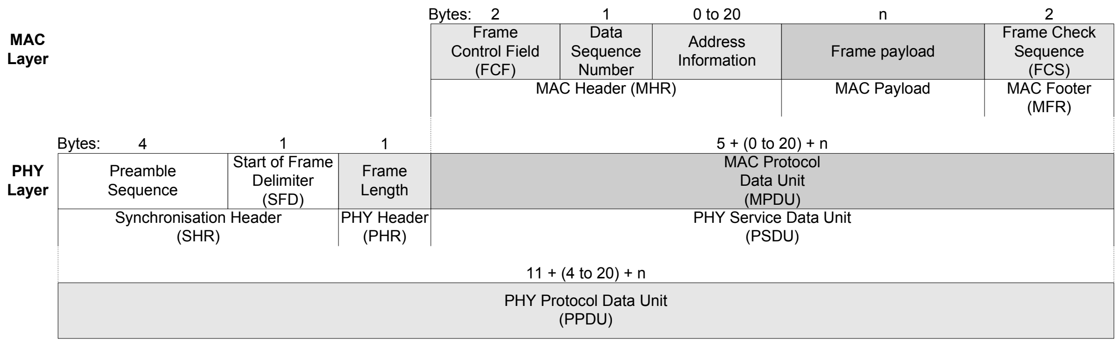


Figure 3-3
IEEE 802.15.4 Packet format [19]. TinyOS uses this structure in order to benefit from the address recognition feature of the CC2420.

3.3 Measurements

3.3.1 Time and Current

The intention of the following measurements is to get an idea about the timing behaviour and the power consumption of the Tmote Sky while sending and receiving packets with the CC2420. The results should provide a base for later implementation of a network protocol. All current measurements represent the power consumption of the whole system, i.e. the Tmote Sky node.

3.3.1.1 Static Current Measurement

The first experiment was made with an ampere meter, so it was only possible to get static data. The goal of those measurements was to get the power consumption of the different low power modes of the microcontroller. For this purpose, we loaded four different programs onto the Tmote Sky. The first one executed a loop. The other three explicitely put the microcontroller into a sleep mode. The node was powered by battery. For measurement, the ampere meter was put in series between the batteries and the power connector of the Tmote. The measured values are shown in table 3-3. The measured values are slightly higher than those in the data sheet from

Mode	**Measured current**	**Current by datasheet[16]**	
		nom	**max**
Running	$3.23mA$	$1.8mA$	$2.4mA$
LPM 1	$194\mu A$	N/A	N/A
LPM 3	$38\mu A$	N/A	N/A
LPM 4	$35.5\mu A$	$5.1\mu A$	$21\mu A$

Table 3-3: Current consumption of the Tmote Sky in different low power modes, CC2420 radio module is switched off

Moteiv[16]. Especially the current in LPM 4 is by a factor of seven higher. On older Tmote Sky boards, which are not yet FCC certified, we could measure a current of $7.5\mu A$. Maybe there are some differences in the board design. We could not find the details that were responsible for this effect.

3.3.2 Radio Transmission

3.3.2.1 Time Related

This experiment was intended to measure the duration of important steps in the transmission process over the radio. We wanted to know, how long it takes until a packet's SFD is sent after the send process had been initiated. Additionally, also the arrival time of the packet at the receiver was of interest.

The setup is as shown in figure 3-4: We placed two nodes, one sender and one receiver. They are connected with a wire. With this connection it was possible to measure time relation of sender and receiver. For timing measurement the internal timer on the node was used, which was clocked by the real time clock at 32.768 kHz. For evaluation, each node was connected to a personal computer via USB.

The testing behaviour is as follows:

1. The sender wakes up regularly by a timer and initiates the transmission of a short packet.
2. The radio chip CC2420 generates an interrupt as soon as the start frame delimiter (SFD) of the packet is sent.
3. At the same time, the SFD interrupt signal is transmitted over the connecting wire and generates an interrupt in the receiver node.
4. When the receiving radio module detects an SFD of the incoming packet, an interrupt is signalised.
5. When the whole packet has arrived, the TinyOS network stack signals an event to the higher application layer.

The timestamp of each interesting event (timer, SFD sender, SFD receiver) was sent to the PC. In order not to influence the timing behaviour of the transmission, the

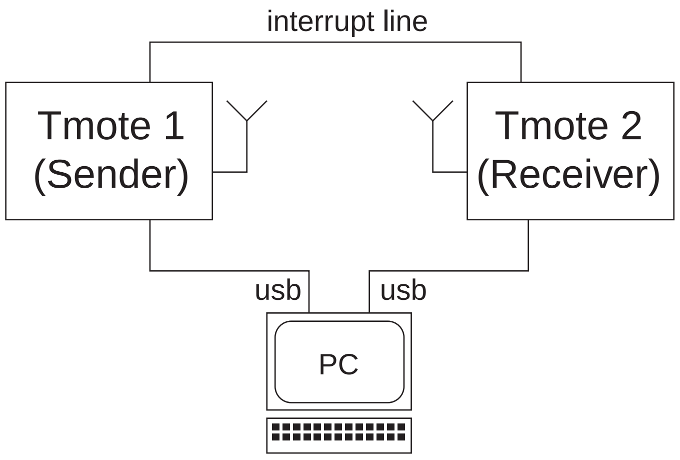


Figure 3-4
Setup of the time measurement experiment. The two nodes are connected by a wire in order to correlate the time, when a packet was sent, on both nodes. For data evaluation, the nodes were connected to a PC.

logging component was implemented as a task. Hence, time critical elements that were implemented as asynchronous code, could interrupt this task. This gives the logging component implicitly the lowest priority.

First measurements showed that the interval between the timer event and the SFD event of the sender is not deterministic (Figure 3-5). This can be traced back to the

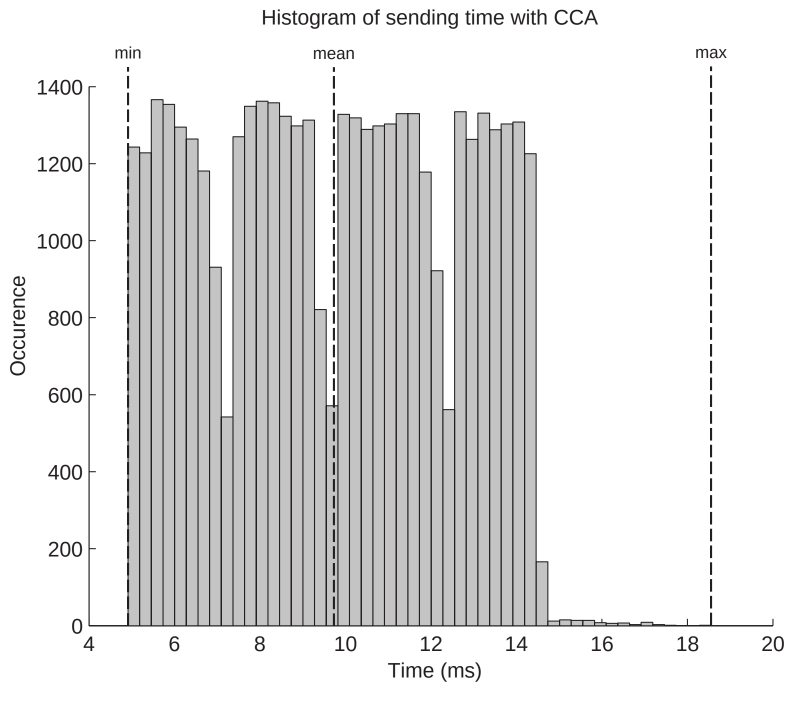


Figure 3-5
Time for sending a packet, including radio startup time

implementation of the MAC-Layer in TinyOS 2.0. By default, carrier sense is done before sending a packet. If the media seems to be occupied, the packet is held back for a random backoff time. Without this feature, the intervals are constant. As could be expected, both SFD events (measured at the sender and receiver) occur at the same point in time or in any case no time difference can be measured with this time resolution. When setting the sender node in sleep mode (radio off) between sending packets, an additional wakeup period of 88 clock ticks is needed. As seen in later experiments, most of this additional time is spent for starting the oscillator of the radio module.

3.3.2.2 Power Consumption

For power consumption estimations we need an exact picture of the current flow in the different transmitting states of the Tmote Sky. This information was gathered by measuring with an oscilloscope (Tektronic TDS 540). In the test setup we had again two nodes, a sender and a receiver. The sender did periodically wake up and send a packet. After that, it immediately went to sleep again. This time, the CCA feature of the radio module was disabled. The relation between the measured current curve and the events that happened in the running program was established by setting IO pins. At every event of interest, one line of code was inserted to set a pin. In this way it was possible to trigger the time of occurrence very precisely with the oscilloscope. As the oscilloscope could only measure voltage, the used current was indirectly measured with the voltage over a 10 Ohm shunt resistor. Data was sampled at a time resolution of $20\mu s$.

3.3.2.2.1 Sender

Figure 3-6 show the resulting oscilloscope output for the sender. The trigger events are indicated by dotted lines.

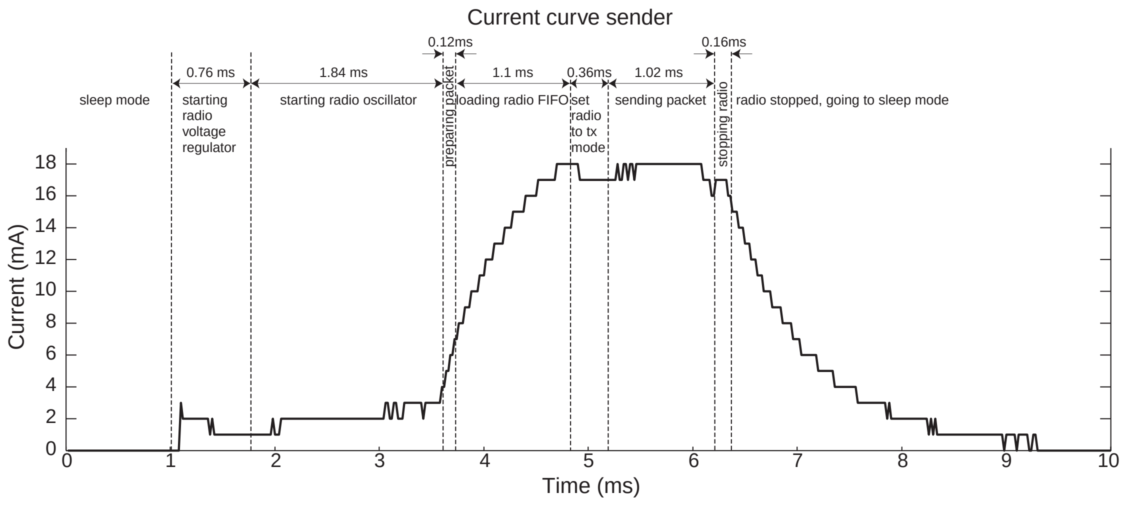


Figure 3-6
Current curve of the Tmote Sky node when transmitting a packet. The time of the events are measured by triggering on IO pins.

Following processes can be observed (corresponding figure tags in brackets):

1. The node is in a low power mode (sleep mode)
2. When it wakes up, it first starts the voltage regulator of the radio module (starting voltage regulator)
3. As the voltage level is stable, the oscillator of the CC2420 is started (starting radio oscillator)
4. The packet is prepared for sending (preparing packet)
5. The packet to send is loaded into the FIFO buffer of the radio module (loading radio FIFO)
6. The radio module switches to transmission state and begins to send the SFD (set radio to TX mode)
7. The whole packet is transmitted (sending packet)
8. The radio module is stopped (stopping radio)
9. The whole system is going to shut down (radio stopped, going to sleep mode)

The round ascent and descent of the curve around the transmission result from charging and discharging capacitors on the Tmote Sky. The sending time interval from radio start until sending of the SFD approximates the lower bound of figure 3-5.

3.3.2.2.2 Receiver

For a complete power estimation, we made the same current measurement for the receiving node. As the receiver does not know the exact arrival time of a packet, we implemented a program that runs in an idle listening mode until the packet arrives. After the full packet has been received, the node goes to sleep. The resulting current

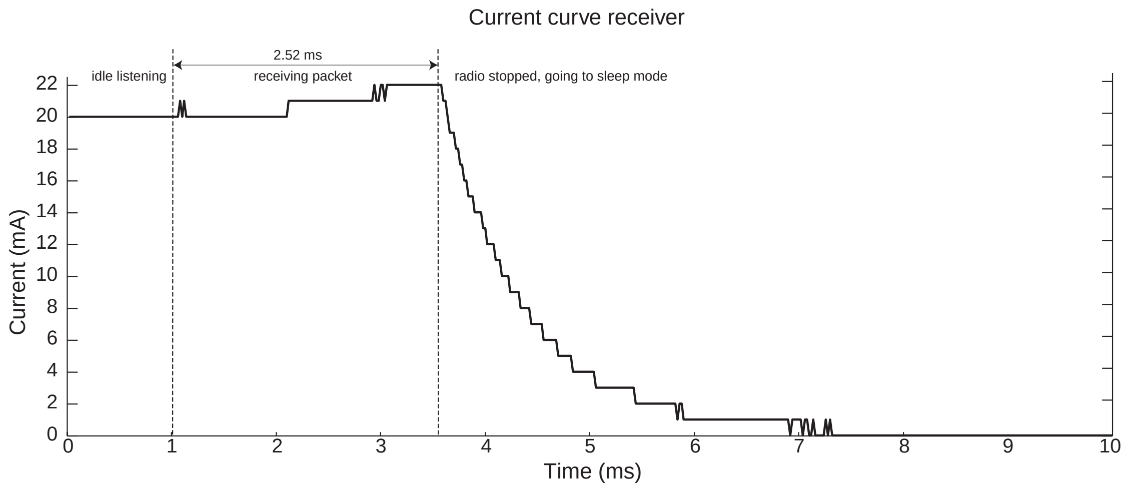


Figure 3-7
Current curve of the Tmote Sky node when receiving a packet. The time of the events are measured by triggering on IO pins.

curve is plotted in figure 3-7.

Following events happened:

1. The node has started and is ready to receive packets (idle listening)
2. A SFD is detected and receiving starts (receiving packet)
3. The whole packet has been received and moved to the microcontroller's memory for further processing. The system is going to sleep mode (radio stopped, going to sleep mode)

As described in the datasheet of CC2420[18], current consumption in receiving (or listening) mode is higher than in sending mode.

3.3.3 Lifetime Estimation Model

With the previously gathered data, it is now possible to create a model for power usage and energy consumption. Also an estimation for lifetime can be made. We divide the phases of a node's program as follows:

Phase	**Duration**	**Current**	**Description**
Startup	$T_{startup}$	$I_{startup}$	Wake-up phase from sleep mode
Shutdown	$T_{shutdown}$	$I_{shutdown}$	Phase for going to sleep mode
Idle/Rx	$T_{idle/rx}$	$I_{idle/rx}$	Total time spent in receiving mode (including idle listening)
Tx	T_{tx}	I_{tx}	Total time spent in trasmission mode (including rx/tx switch)
Sleep	T_{sleep}	I_{sleep}	Low power mode, radio module off

Table 3-5: Parameters of the power estimating model. All currents are average values.

For a program with a periodic behaviour, the mean current consumption can be calculated as follows:

$$I_{avg} = \frac{\sum_{phases} I \cdot T}{\sum_{phases} T} \tag{3.1}$$

Based on this mean current and with a given battery capacity Q_{bat}, a lifetime estimation can be made:

$$T_{lifetime} = Q_{bat}/I_{avg} \tag{3.2}$$

Different implementations have different time characteristics. Duty cycle and the ratio between sending time and receiving time differ. $T_{startup}$ and $T_{shutdown}$ can be assumed as constant. Those two values and the current usage in each phase can be taken from the previous measurements. Repeatedly measured curves were almost identical, so we took the average current and time over ten measurements. The values are shown in table 3-6. We use this model in chapter 5 in order to characterise our implementation.

Parameter	**Value**
$T_{startup}$	$3.804ms$
$I_{startup}$	$5.5mA$
$T_{shutdown}$	$3.008ms$
$I_{shutdown}$	$4.6mA$
$I_{idle/rx}$	$20.8mA$
I_{tx}	$17.5mA$
I_{sleep}	$35.5\mu A$

Table 3-6: Values of the model. Each is an average of measured values

4

Connection to the Deployment Support Network

4.1 Introduction

Developing applications for wireless sensor networks implicates various difficulties. Not only that it is hard to debug an embedded system but also applications run on several sensor nodes, maybe up to a few hundreds, which makes the task of finding an error even harder. The reprogramming of a whole network in is very time-consuming when each node needs to be programmed individually and such code updates are common. To make development of big sensor networks feasible, a system is required that (1) can handle code updates efficiently, (2) is able to monitor the state of every node and (3),for testing purposes, is also capable of sending commands to arbitrary nodes to influence their behaviour. For all these tasks, information needs to be exchanged between the sensor network, later referenced to as target network, and a host. This can be done in several ways.

4.2 Deployment Support

4.2.1 Communication over the Target Network

Additional messages for these tasks could be sent over the radio interface of the sensor nodes. Deluge [24] for example provides a network programming feature based on target network communication. Such a solution assumes that the sensor application already has a working network protocol stack, which is not always the case. In fact network protocols are a main area of interest when it comes to wireless sensor networks, so many applications will contain new and untested implementations of a network stack. Giving a guarantee for the robustness of such a network in development is almost impossible. Another fact that should be considered is, that additional messages would most certainly influence the behaviour of the whole network. With a contention based protocol for example, there would occur more collisions and messages would have longer delay. One can also think of a scenario where the bandwidth of the network is too small for any additional debug packet. Software updates need additional memory to save the received update before it is flashed. Some nodes may not have this additional storage.

4.2.2 Communication over a Wired Overlay Network

Another way to get the debugging information from the sensor network is to use a second network infrastructure. There are testbeds [25][26] that set up a backbone based on Ethernet or WLAN (802.11b) to connect several base stations. These stations could be normal personal computers or dedicated hardware like Stargate[27]. Every base station has access to several nodes via a wired interface. Other testbeds [28][29] mount an Ethernet adapter to every target node and connect them with cables. Such networks need a huge infrastructure, especially when it comes to widely distributed systems with a bigger number of nodes.

4.2.3 Communication over a Wireless Overlay Network

Yet another option is an entirely wireless overlay network. Such a network is being developed at ETH. This Deployment Support Network (DSN)[30] is based on BTnodes [31] that maintain an independent multihop ad-hoc network. Every sensor node of the target network is attached to a BTnode via a target adapter. The target network and the DSN are coexisting as illustrated in figure 4-1. All debug infor-

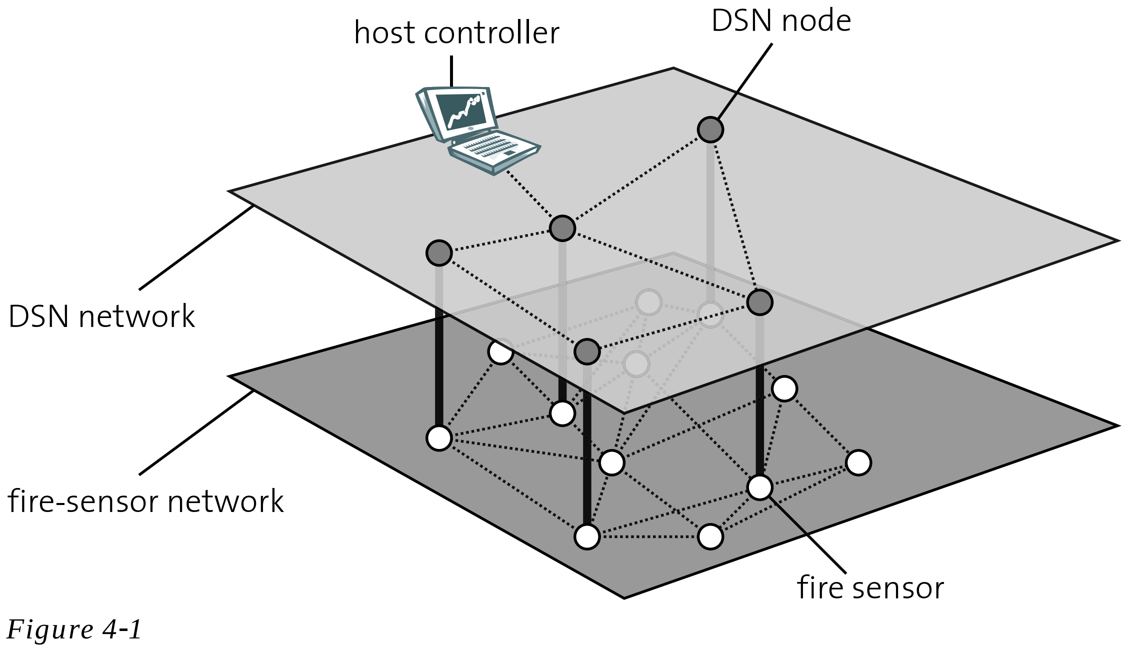


Figure 4-1
DSN with targets. The Deployment Support Network acts as separate Network to gather status information that can be used for debugging. The network can also be used for surveillance and code updates.

mation and code updates are transported over the DSN, not influencing the target network. Every platform needs its adapter in order to be supported by the DSN. This adapter consists (1) of the target specific implementation in the DSN software, (2) a hardware connection between DSN node and target node and (3) an implementation on the target node for communication with the DSN (sending of log date, receiving of commands). At this time, the DSN has target implementations for the following motes: BTnode rev3[31], Shockfish TinyNode 584[32] and Siemens A80.

The last solution for deployment support has two advantages over the testbeds mentioned before: There is no need for long cables, which makes this DSN more flexible. Furthermore the targets can be placed virtually everywhere. This feature allows the nodes of our wireless fire sensor network to be placed in the same locations as for conventional fire sensors. For this reasons, we used this DSN for our master thesis.

4.3 Target Connection for the Tmote Sky

There existed no target adapter for the Tmote Sky, so we had to design it first. Following functionality is implemented in the DSN and should also be applicable for the Tmote Sky target:

- Resetting
- Target programming
- Communication in both directions
- Target powering (optional)

In the remaining chapter, we describe the challenges and the details of the target adapter implementation for the Tmote Sky.

4.4 Target Programming

4.4.1 Challenges

There are three options of how to program the TI MSP430 microcontroller [17]. Either via the default bootstrap loader, the JTAG[1] interface or with a new implementation of a bootstrap loader. A short overview on this topic is given in table 4-2.

4.4.1.0.3 Bootstrap Loader

From the DSN point of view, there exists already an implementation for the normal MSP430 bootstrap loader protocol [33]. The Tmote Sky has an onboard USB interface that is connected to the pins of the bootstrap loader (BSL) of the microcontroller (Figure 4-5). This makes programming very convenient if the node is directly connected to a personal computer, as no additional hardware is required. The fact that the USART1 module of the MSP430 is also wired to this USB connection, makes serial communication available through the same interface. This USB feature makes

[1]An acronym for Joint Test Action Group, is the usual name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards using boundary scan.

Programming method	**Advantages**	**Disadvantages**
Bootstrap loader	Default programming procedure Pins for BSL and UART1 wired together on Tmote Sky. Needs only one connection for programming and communication	Pins are connected to the onboard USB interface (not easily reachable). Need of a USB master circuit on the DSN side
JTAG	Default programming procedure Pins are accessible	Needs additional protocol implementation on the DSN Needs an additional interface for communication
Adapted bootstrap loader	Can use arbitrary default communication interface (USART0, USART1) Can use same pins (e.g. only one connection) for programming and communication	Needs special programming procedure

Table 4-2: Comparison of different programming methods for the MSP430

the connection to the DSN more complex. A simple direct connection to the pins of the bootstrap loader is not possible. To use this interface all the same, it would require designing an additional adapter board with a USB master on it.

4.4.1.0.4 JTAG

For the second programming option, the Tmote Sky provides a connector to the JTAG interface of the MSP430, but this requires an implementation of the JTAG protocol in the BTnode DSN software. It would also be a big effort to generate a proper software based clock signal for JTAG. Moreover, this interface is not intended to be used for communication of the running program and therefore an additional connection would be necessary for communication between BTnode and Tmote Sky in order to exchange logging information with the DSN.

4.4.1.0.5 Adapted Bootstrap Loader

Most freedom is provided by a new implementation of a bootstrap loader. This solution uses the feature of the TI microcontroller, that a user program has full access to the internal flash memory. Such custom programmers exist already for wireless sensor platform applications like Deluge[24] or JAWS[34] which provide an in-network programming feature. We decided to use this method because it lets us utilise an easy accessible interface (UART0) for both, programming and communication with the DSN. The pins of this serial interface are provided by the 10 pin expansion connector of Tmote Sky. While using this interface, it is possible to build an adapter that adds as less additional hardware as possible to the existing DSN. Our adapted bootstrap loader would have to implement code reception over the serial interface, preferably compliant to the MSP430 bootstrap loader protocol. Then it would have to save the received program into the flash memory.

4.4.2 Realisation of the Bootstrap Loader

Our new bootstrap loader is actually a normal TinyOS program, running on the microcontroller. Like the default bootstrap loader, it can be invoked by a special entry sequence on a defined pin. Additionally is loads program code over the serial interface UART0 and saves it into the flash memory.

As the new BSL resides in the same memory area like other programs, we have to take care, that the new BSL does not disturb the execution of any other program. Special attention has to be paid on interrupts.

4.4.2.1 Memory Organisation of the MSP430

As depicted in figure 4-2, the flash memory of the MSP430 is partitioned into segments of 512 bytes. Single bits, bytes or words can be written to flash memory, but the segment is the smallest size of flash memory that can be erased.

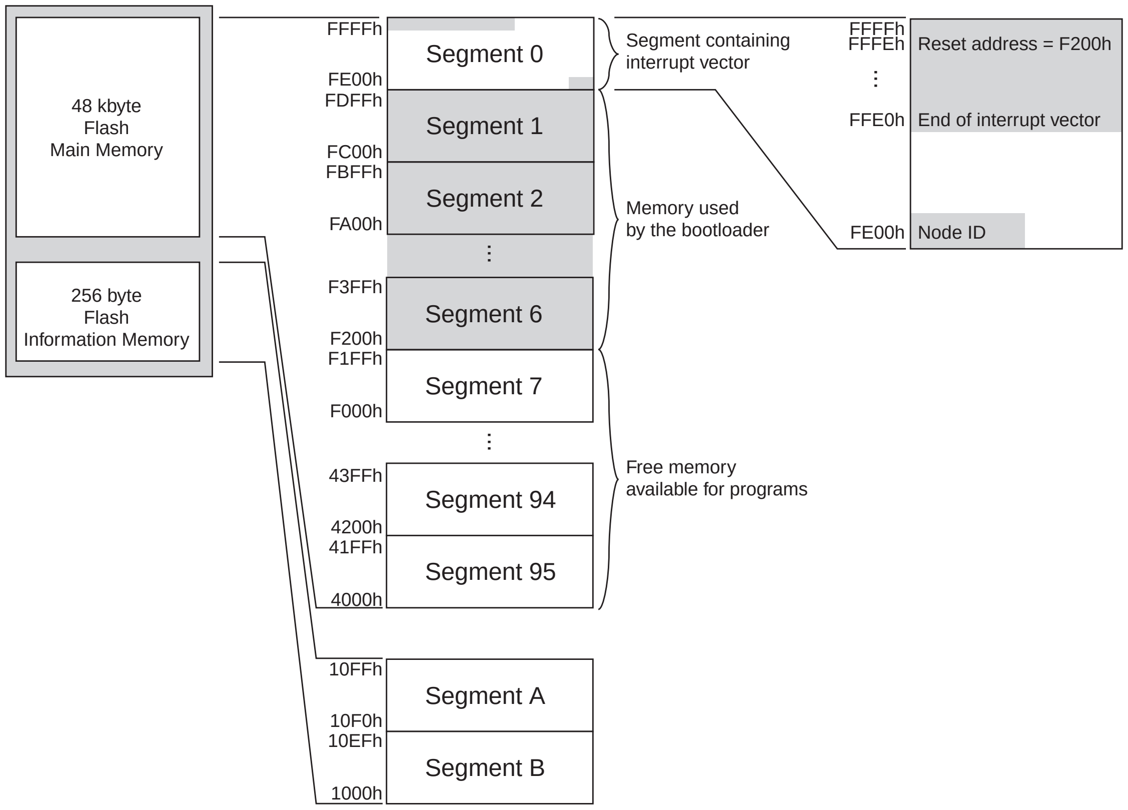


Figure 4-2
Memory organisation of the MSP430F1611. The bootloader is once programmed into the upper end of the memory. The interrupt vector resides at FFFFh.

Program code usually starts at the lowest flash memory address (4000h, Segment 95) and is contiguous. Additionally, every program saves its interrupt vector in segment 0 at the end of the memory block. Every time the microcontroller starts, it first reads the reset address out of the interrupt vector. Then the program counter is loaded with this value and code execution begins.

4.4.2.2 Location of the Adapted Bootstrap Loader

For convenience, a user should not need to compile his program in a special way in order to use our BSL, so we assume the normal properties of a program like described in the previous section. In order to reserve as much memory as possible for a user program, we place the adapted BSL at the top of the memory, right before segment 0. Segment 0 can not be used, because the interrupt vector is overwritten with every new program and, as mentioned before, in this cycle the whole segment 0 has to be erased. Now the maximal space available for another program is reduced by the size of the new bootloader (approx. 4kBytes), but programs can still be as big as 45.5kBytes and do not need to be compiled with special displacement in memory.

4.4.2.3 Interrupts

In segment 0 there is only one interrupt vector. The bootstrap loader has to guarantee, that every time another program is running, the right interrupt vector is in place. Otherwise there would be a conflict and the behaviour of the program would be incorrect, every time an interrupt is generated.

We managed to implemented the new bootloader without the need for interrupts. All interrupt routines could be replaced by polling registers. This allowed us to disable all interrupts when running the bootloader, making it independent of the interrupt vector. Data of another program in this table does not affect the behaviour of the bootloader.

4.4.2.4 Start-up

When the adapted BSL is installed, there are actually two different programs loaded into the MSP430 but the controller can only start one. To solve this problem, we start the BSL first and decide, based on the state of an input pin (Figure 4-3), whether to start the bootloader or the user program. This behaviour is sketched figure 4-4. To maintain this starting order, the adapted BSL always makes sure, that the reset vector in the interrupt vector always points to the start address of the bootloader.

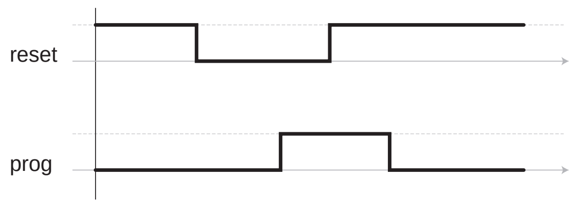


Figure 4-3
Bootstrap loader entry sequence

4.4.2.5 TinyOS Node Address

In TinyOS, the identification number of every node is saved in the program code at a non deterministic memory address. This implies that every node programmed with the same binary would have the same identifier, which is the case when code is distributed and programmed by the DSN. To circumvent this issue, we assume that the adapted BSL on every node is programmed with a unique ID. Whenever the adapted BSL is started, it sets its own node ID to a fixed address in segment 0. If the user program is compiled with the NesC DSN component, then this component initially reads out the ID at this address in segment 0 and sets it as its own network identifier.

4.4.2.6 Program Flow

The chart of the program flow is shown in figure 4-4. The BSL entry sequence at start up lets the bootloader execute the left branch. This branch is essentially the TI BSL protocol with two exceptions. A mass erase command does erase all segments but the bootloader segments and every time segment 0 is erased, the bootstrap loader rewrites its start address into the reset vector. This ensures that the bootstrap loader is always started when the microcontroller starts. Also shown in the flow chart are the handling of the TinyOS network identifier and the start-up sequence as described in the previous sections.

4.4.3 Installation of the Bootloader

The DSN-bootstrap loader is installed with the normal procedure over the USB-Interface. The node keeps the ID that was declared while installing the bootstrap loader. The command to install the bootloader looks like this, where ID is the node identifier:

```
make tmote install,ID bsl,COMPORT
```

The bootloader will be removed if a real mass erase is performed. This is usually the case when programming with the default BSL or over the JTAG interface.

4.5 Communication

4.5.1 Challenges

When the program on the target node is running, it sends its logging date to the DSN. The standard (an only implemented) method in JAWS[2] is to send those messages over an UART interface. As shown in figure 4-5, there are two interfaces that are intended to be used for this kind of communication: USART0 and USART1. As mentioned earlier, access to the latter module is very difficult due to the USB controller. Therefore we use the former interface (USART0) for this functionality, the same as already used for programming.

Tmote's Chipcon 2420 radio module is connected to the same USART0 hardware module. The USART0 can be configured exclusively in SPI, I2C or UART mode.

[2]Name of the software that is running on the BTnodes of the DSN. Jaws is an allusion of several (Blue)teeth building a jaw.

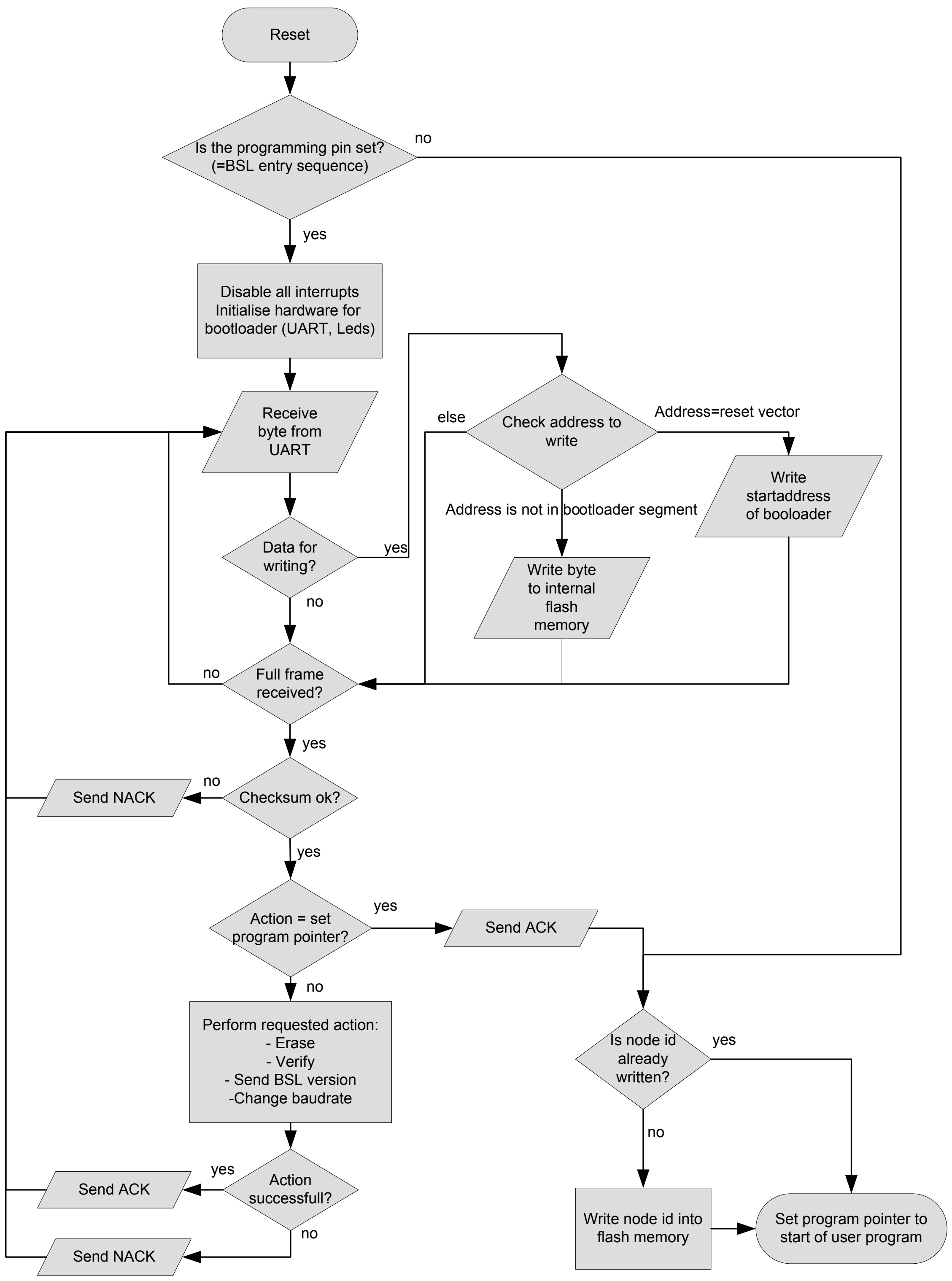

Reset
Is the programming pin set? (=BSL entry sequence)
no
yes
Disable all interrupts Initialise hardware for bootloader (UART, Leds)
Receive byte from UART
else
Check address to write
Address=reset vector
Write startaddress of booloader
Address is not in bootloader segment
Data for writing?
yes
no
Write byte to internal flash memory
no
Full frame received?
yes
Send NACK
no
Checksum ok?
yes
Action = set program pointer?
yes
Send ACK
no
Perform requested action: - Erase - Verify - Send BSL version -Change baudrate
Is node id already written?
yes
no
yes
Send ACK
Action successfull?
no
Send NACK
Write node id into flash memory
Set program pointer to start of user program

Figure 4-4
Flowchart of the bootstraploader for Tmote Sky.

Each mode uses different pins on the MSP430. The radio module is connected in SPI mode and communication to the DSN node is done in UART mode. This implies following restriction: Communication can either go on between the microcontroller and the DSN (via expansion connector and UART) or between the microcontroller and the radio module. The program running on the Tmote Sky must ensure, that the USART0 module is in the right mode for the ongoing communication. Implementation details are provided in the following paragraph.

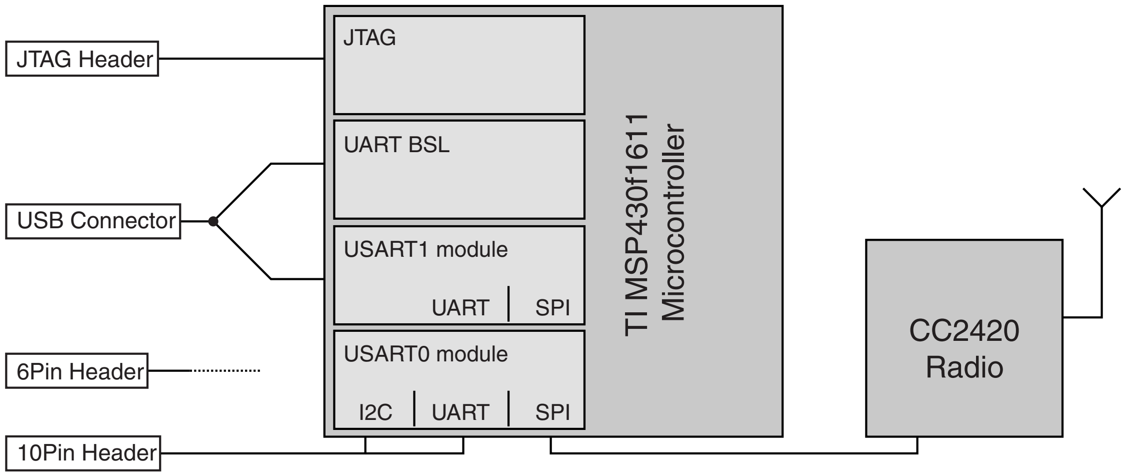


Figure 4-5
Available connectors to the Tmote Sky. USART0 and USART1 provide several operation modes but only one at a time. The radio module is hard wired to the USART0 module in SPI mode.

4.5.2 DSN Component

We have written a TinyOS component that can be wired to any other program in order to use the DSN. This component properly initialises the UART and handles the communication with the DSN-node (logging and receiving commands). As described in chapter 3, shared resources (UART and SPI) are handled with arbiters in TinyOS 2.0. Components that want to use a shared resource first have to request it and wait then until access is granted. This works fine when sending log messages to the DSN. Receiving commands from the DSN node requires an additional mechanism that tells the DSN component when to acquire the USART0 module, since the UART cannot be used when the radio module is in use.

This is accomplished using a RTS/CTS scheme. Every time the DSN node wants to send data to the Tmote target, it sets the RTS signal on a pin and waits until the Tmote node has successfully configured the UART module. Then the target notifies that it is ready via the CTS signal. Since the RTS signal triggers an interrupt on the Tmote Sky, the microcontroller can respond even if it was in sleep mode.

4.5.2.1 Features

Following events and commands are provided by the DSN interface:

```
interface DSN {
        command error_t log(void * msg);
        command error_t logLen(void * msg, uint8_t len);
        command error_t logError(void * msg);
        command error_t logWarning(void * msg);
        command error_t logInfo(void * msg);
        command error_t logDebug(void * msg);

        async command void logInt(uint32_t n);

        command error_t logPacket(message_t * msg);

        command error_t stopLog();
        command error_t startLog();

        event void receive(void * msg, uint8_t len);

        command void emergencyLogEnable(uint32_t timeout);
        command void emergencyLogDisable();
        command error_t emergencyLogAdd(
                                void * pointer,
                                uint8_t numBytes,
                                uint8_t * description);
}
```

Listing 4.1
Code: Interface to the DSN component

We provide seven commands for logging messages. Two of them are for general purpose. `log()` takes a null terminated string and `logLen()` a string with known length. The other four logging commands correspond to the log levels defined in JAWS, each representing a level of severity, listed in table 4-3. Numbers are logged

Log Level	**DSN Command**	**Description**
LOG ERROR	logError(void * msg)	Severe errors that impact the system.
LOG WARNING	logWarning(void * msg)	Warnings. Indicate a non-critical failure.
LOG INFO	logInfo(void * msg)	Informational messages.
LOG DEBUG	logDebug(void * msg)	Debug output intended for developers.

Table 4-3: Log levels of JAWS [34]

with the `logInt()` command. Like standard C, the DSN component replaces `%i` in `msg` with the previously logged integers. Other replacements are shown in table 4-4.

The logging of numbers in messages is therefore done in several consecutive `logInt()` commands followed by one `log()` command. This splitting is necessary because NesC does not provide functions with unknown number of arguments. For

Symbol	Replacement
%i	decimal representation
%h	hexadecimal representation
%b	binary representation

Table 4-4: Symbols for numerical logging

logging of radio packets in TinyOS, we provide the logging function `logPacket()`, which writes the header and payload of the packet in hexadecimal format to the UART.

A TinyOS component that wants to evaluate received commands from the DSN needs to implement the event `receive()`. As soon as a whole command is received, the event is signalled.

There are testing scenarios where we are not interested in the log messages when the system behaves normal, but in the messages of an error case. If this error does prevent the program flow in such a way, that normal logging output is not possible, we are at least interested in the state of the program. The emergency logging feature lets the user specify a number of variables that are logged in an error case. The error case can be defined by a timeout, in which no logging activity is done.

The two commands `logStart()` and `logStop()` are provided to control the active logging output. If it is disabled, logging is only done to a local buffer. This can be used for instance, when the message is logged in a time critical part of the program. In this case, the message is just buffered, which need less computation power. This controlling of logging output can also be applied to make sure that the USART module is not used for logging at a specified moment. Critical radio commands like the modification of the transmission buffer make this necessary.

4.6 Tmote Sky Target Adapter Hardware

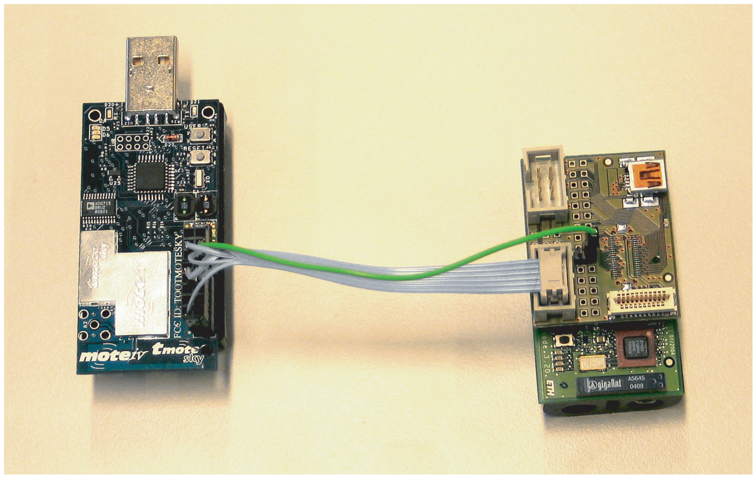
The hardware of the target adapter consists of a 7-wire cable and the DSN adapter board for BTnodes. Tables 4-5 and 4-6 show the connected signals.

Target Logic	Target Pin	Tmote Net	BTnode J1	AVR Pin	BTnode Net
TG_RESET	6	RESET	26	PB1	SCK
TG_PROG / TG_RTS	3	GIO2	27	PB0	SS
TG_CTS	4	GIO3	29	PE6	PE6

Table 4-5: Tmote 6-pin Expansion (U28)

Target Logic	Target Pin	Tmote Net	BTnode J1	AVR Pin	BTnode Net
ISP_MOSI_TXD	2	UART0RX	4	PE1	UART0_TXD
ISP_MISO_RXD	4	UART0TX	5	PE0	UART0_RXD
TG_BAT_SENSE / VCC_TG	1	AVCC	29	PF6	TDO
GND	9	GND	40	GND	GND

Table 4-6: Tmote 10-pin Expansion (U2)



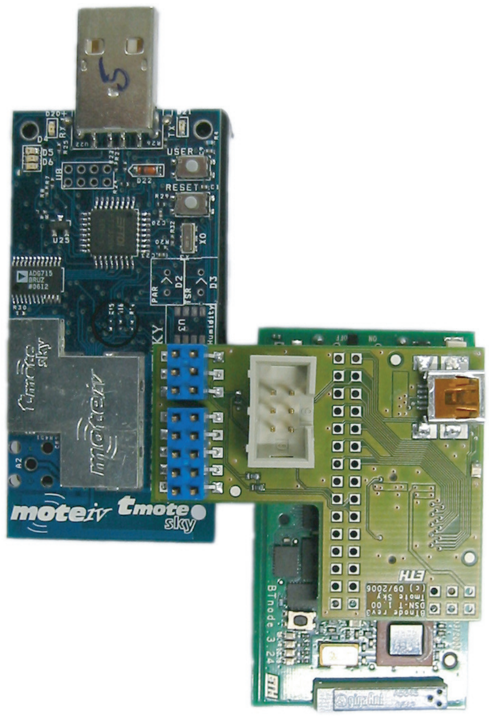


Figure 4-6
Fotos of the Tmote Sky DSN-Adapter. The prototype (at the top) used a ribbon cable and the BTnode DSN adapter board. The final Tmote adapter board (below) has a special connector in order to fit to the displaced jumpers of the tmote Sky.

5 Implementation of Status Monitoring

In this thesis, we focused on the monitoring of a sensor network, especially on the detection of failed nodes. The goals were to design and implement a robust system which consumes very little energy.

5.1 Failure Detection Strategies

Following possible failures can prevent a sensor node from providing its duty:

- The battery of a node has drained out.
- A hardware defect occurs.
- A software failure occurs that makes the behaviour of the node unpredictable.
- The needed communication channels to a node are permanently missing or disturbed.

The common characteristic of all cases is, that the missing node is unable to send proper radio messages. Obviously, a node can not detect its failing by itself, so neighbours have to find out somehow the erroneous state of the node. When they eventually have found, that a node is missing, this can be reported to a base station. With this method, only missing nodes are reported and the network is supposed to run fine, if no information is provided. This would actually work, if the network is highly reliable and no message losses occur. In WSNs there are often lost packets due to interference, packet collisions or limited transmission power. A sent failure report can therefore be lost and the absence of information could also indicate a network failure.

Another approach is to positively report error-free nodes like a network heartbeat. In this scenario, the base has to know all available sensors in the network. Different to the error reporting approach, missing nodes can be actively searched by the base station. Both failure detection approaches referenced in section 2.2 are based on such a heartbeat. Each node periodically sends a status message and as long as this information reaches the interested nodes or the base station within a predefined interval, the referred node is considered as alive.

5.2 Approach

Our status monitoring is based on the information of active nodes. In steady state, status information of all sensor nodes is gathered towards a base stattion within a *reporting wave*. In this wave, information is aggregated and routed through various paths to achieve robustness through redundancy. At the base, the reported status of the network is acknowledged with an *acknowledge wave* in the opposite direction. The co-ordination of these waves is done at the base station with a time division multiple access (TDMA) schedule. Every node gets a specified sending slot in order to avoid packet collisions. Synchronisation and schedule information are propagated within this acknowledge wave. At start-up, every node registers itself at the base station.

For later usage, we call the combination of the *reporting wave* and the *acknowledge wave* a *round*. A *period* starts, when all nodes switch from the low power mode to the active mode for sending a *reporting wave*.

5.2.1 Requirements and Assumptions

The implemented monitoring sytem has to satisfy following requirements (Problem task, Appendix A):

1. Failed nodes have to be reported by the whole system in a hard time limit $T_{Report,Max}$ of 100 seconds (*Time constraint*)
2. Robustness: Individual temporary link failures must not affect the reporting facility (*Robustness*)
3. If one radio frequency is temporary disturbed, an alternative frequency must be used for proper functionality (*Robustness*)
4. The nodes have limited energy and should therefore mostly stay in a sleep mode (*Low power*)

For the following considerations we assume, that the network is bounded to N nodes. Additionally, every node is no further away than H hops from its base station, which implies, that the network has a maximum diameter of $2 \cdot H$ hops. Not every node might have a direct link to the base station, so multi-hop functionally has to be implemented. When deployed, the position of every node can be chosen within a certain radius in order to increase the probability of having a connected network. Once put in place, nodes are not moved around anymore. Therefore we do not have to consider link quality changes due to mobility but rather due to interference from electrical devices such as communication systems (e.g. 802.11 or Bluetooth) or household appliances (e.g. micro wave ovens).

5.3 Concepts

In this section, we describe the strategies that are used in our implementation in order to meet the requirements for robustness, time constraint and low power.

5.3.1 Robustness

5.3.1.1 Redundancy and Acknowledgements

There are two main concepts to increase robustness of communication: Acknowledged messaging and redundant message paths. We use both principles in for our reporting system.

To achieve redundant communication, we exploit that transmissions from a node in a WSN can be overheard by all its neighbours. If we send the alive information in local broadcasts (to all 1-hop neighbours), every listening node can aggregate the status of its neighbours. For information aggregation we use a bitmask. Such a strategy is comparable to flooding. It gradually enhances the robustness of the network but also increases the power consumption. Link failures or total loss of communication can influence the completeness of both waves. Although there is redundancy in the information flow, it can happen that a node misses a message. For this case, our monitoring system implements a recovery mechanism.

We differentiate two cases:

1. The reporting wave does no contain the information of all registered nodes
2. A node does not receive the acknowledge wave and is therefore not synchronised anymore

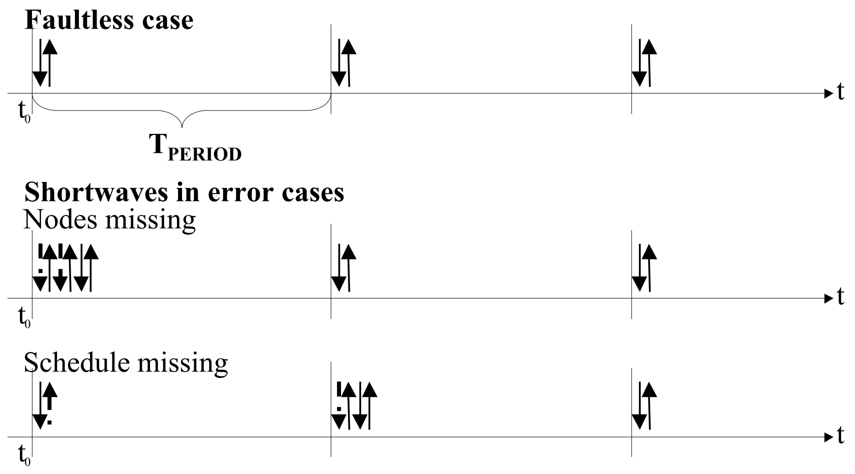


Figure 5-1
Overview on the recovery mechanism. In the faultless case, only two waves are sent in one period.

When a node is not reported in a round, a rerun of the waves in a *recovery round* could bring the missing information. This is the case, when this failure is caused by a temporary link error or an unsynchronised node. In our case, the sink sets a flag in the acknowledge packet that indicates, that an immediate next round follows after the regular acknowledge wave. This behaviour is shown in figure 5-1. The following wave rounds end either when all nodes are reported or when the maximum count of retries is reached.

In the case, where an outgoing acknowledge wave is affected by errors (Figure 5-1, last case), some or all nodes did not receive the time synchronisation information. Unsynchronised transmissions can lead to collisions and should be avoided. Nodes that have lost synchronisation do not send a message until they have received time information again. Meanwhile, this node has to sustain its schedule based on the local clock. In order to compensate clock drift, the wake-up phase has to be started earlier, introducing an additional guard time. The successful recovery process of this case is depicted in the last row of figure 5-1.

5.3.1.2 Channel Fault

When an entire frequency is temporarily jammed (e.g. by a garage opener), there could be a whole wave round affected. We address this concern by a simple channel hopping. Every set of recovery wave rounds is sent in an evading channel. Between the two channels used, there are some buffer channels in order to avoid frequencies potentially used or affected by the jammer. With this mechanism, a jammed default channel can be circumvented with two additional short wave rounds in the auxiliary channel. A first round for resynchronisation and a second round for the final reporting.

5.3.1.3 Simple Implementation

We address robustness in terms of 'failure free software' with the strategy of a simple implementation. This keeps the overall state machine of the program manageable and thus reduces the possibility of errors. Furthermore, a simple algorithm needs less computation and therefore possibly less power.

5.3.2 Meeting Time Constraints

The used TDMA scheme gives us the ability to react on missing nodes after a well known time because the duration of each wave, reporting and acknowledging, can be limited to a determined duration. This is done by generating a TDMA schedule in such a way that the sending order of the nodes is according to the logical information flow. If, for the reporting wave, it can be ensured, that outer nodes (bigger hop distance to the base) send before their inner neighbours, the alive information of the whole network can be gathered with only one message per node. The same applies to the acknowledge wave in the opposite order. Unlike contention based protocols, TDMA approaches are inherently collision free. This and the fact, that every wave consists of only one message per node leads to an exactly definable duration for each wave. Our status monitoring system sorts the time slots for the nodes according to their hop distance from the base station. The drawback of such a protocol is its overhead for schedule calculation and the need of a network synchronisation. In section 5.4.2, we will describe our simple synchronisation approach in detail.

As seen in the previous section 5.3.1, retransmissions can occur when not all nodes are reported in a wave round. The base station maintains a node table that contains the time when a node was last reported as beeing alive. When an entry update is missing for more than the maximal reporting time $T_{Report,Max}$, the according node is reported as missing. In order not to report temporarily missing nodes (false positive), more wave rounds can be done per $T_{Report,Max}$.

5.3.2.1 TDMA Scheme

An overview on the principle of our TDMA scheme is given in figure 5-2 (Example for a network consisting of $N = 16$ nodes). Every wave is divided into N slots. At the beginning of a period ($t0$), the reporting wave takes place. In this phase, all nodes are listening for alive packets. Every node sends its aggregated alive information in the assigned slot. When the reporting wave has finished, the sink initiates the acknowledge wave. If all registered nodes are reported, the whole network is put in a sleep mode.

The network schedule is calculated at the sink and then propagated within the acknowledge wave. The order in the schedule is determined as follows: In the reporting wave, nodes with a higher distance to the sink have earlier slots assigned. The ordering between nodes with the same hop count is random. The slots are assigned contiguous from the latest slot towards the earliest. The reporting slot of the sink is not used for transmission but for calculating the schedule and evaluating the received status information.

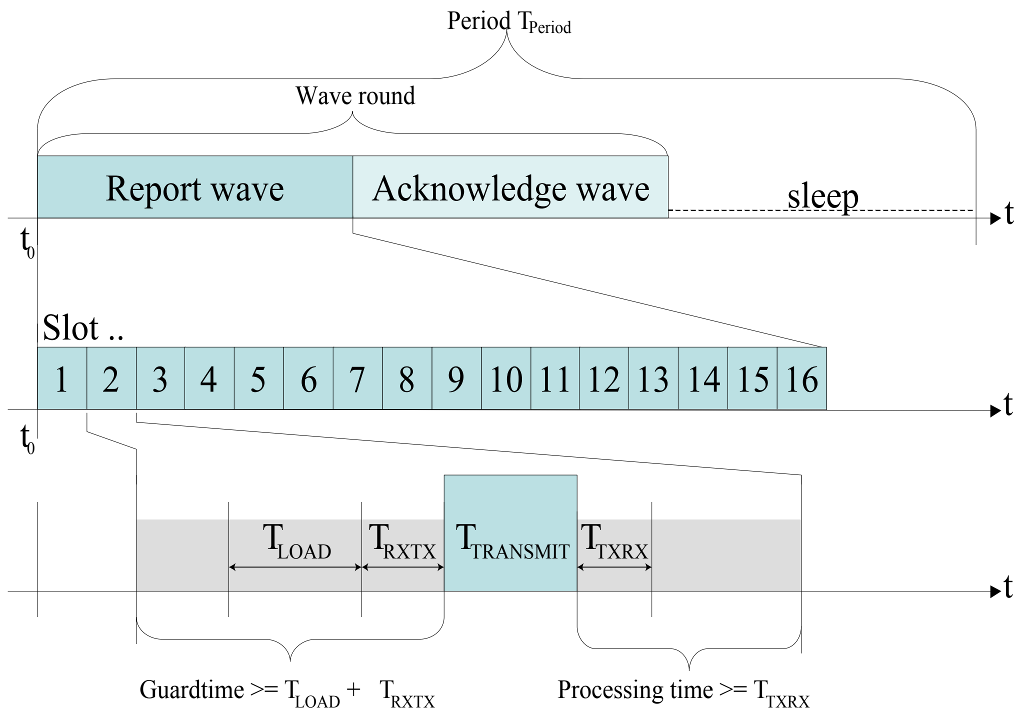


Figure 5-2
TDMA scheme for one wave round. In this example, the network consists of maximal 16 nodes ($N = 16$)

The third axis in Figure 5-2 shows the time composition of a slot. As already discussed in chapter 3, the radio module needs time T_{Load} for loading packets into the FIFO buffer and time T_{RxTx} respectively T_{TxRx} for switching between receive and transmit mode. Additionally, each slot contains a guard time in order to avoid overlapping transmissions due to clock drift. This is mainly necessary in the report wave, as nodes have not synchronised for the duration of a period. If two neighbours are assigned to contiguous time slots, the second node must be able to process the received information from the forerunner. For this purpose we reserve the time $T_{Processing}$ in each slot. The values for all those time parameters are derived in section 5.4.4.

5.3.3 Low Power

We address this concern with the TDMA approach described in the previous section 5.3.2.1. This scheme does not only determine when a node has to send, it also allows to let the sensors sleep, when no transmission is to be expected. In other words, idle listening time is effectively minimised. The schedule also helps to minimise overhearing and avoids collisions, a main cause of power loss in contention based MAC protocols.

5.4 Implementation Details

In this section, we discuss some parts of the implementation of the status monitoring system in TinyOS. This should not be a repetition of the concepts, but rather describe how theoretical work was realised practically. We also present an example of how we used the DSN for the implementation. Our implementation was made for a network with $N = 16$ nodes and a maximal hop count H of 2.

5.4.1 Component Overview

Figure 5-3 shows an overview over the directly used components of the implementation in TinyOS. Every component with a *C* at the end of the name is a configuration that is wired up from several sub-components. The dark components *ClusterNetworkP*, *ClusterAdminC* and *DSNC* contain most of the coding work of this thesis. *ClusterNetworkP* is the main block of the failure detector. *ClusterAdminC* manages the scheduling and *DSNC* represents the DSN component for logging and receiving commands. The components with a dashed background contain the changes made to the CC2420 network stack. At the bottom, there are six components with dashed borders. These are generic components, which means, they represent each an individual instance of the same component. For every packet type (from left to right: Acknowledge packet, report packet, report packet with a joining request), there exist a dedicated sending and receiving component.

5.4.2 Time Synchronisation

Every node has a local clock (calibrated by a crystal). These clocks do mostly not run exactly with the same offset and drift. Clock drift is caused by manufacturing errors of the crystal. Additionally, the clock frequency depends on temperature. In the case of the Tmote Sky, the manufacturing precision of the crystal is 20ppm [44].

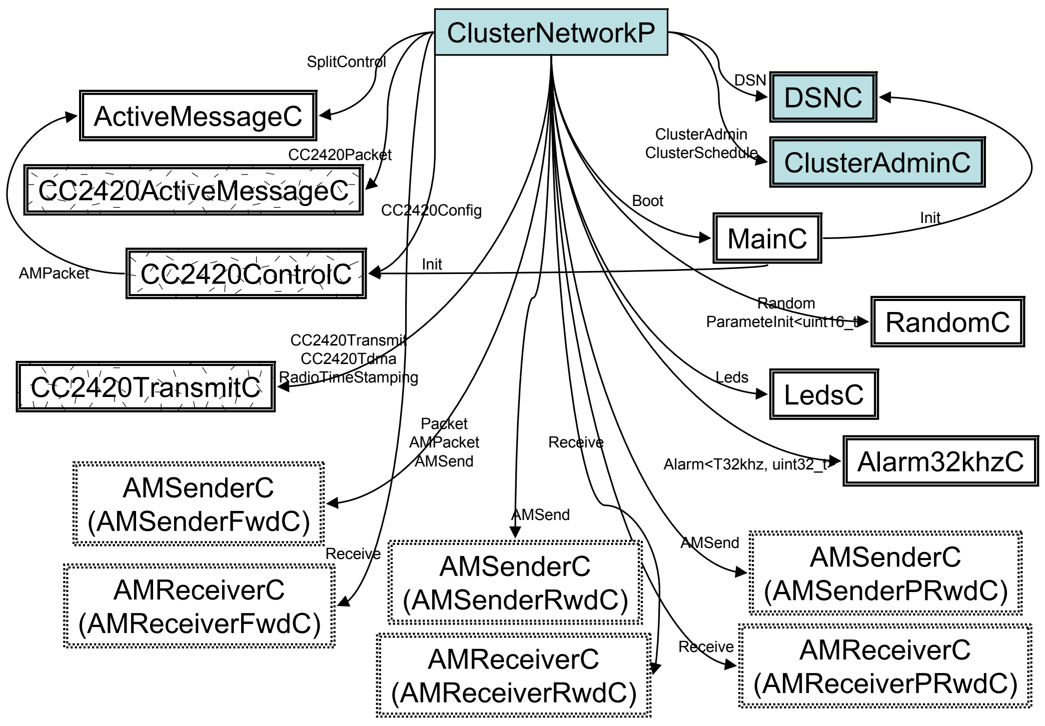


Figure 5-3
Wiring of theimplementaton of the failure detector. Each box represents a component and each arrow a wiring through the indicated interface.

Offset adjustment between two nodes can be done in several ways[40]: unidirectional, with a roundtrip time measurement or with reference broadcasts.

With unidirectional synchronisation, only one packet is sent. The receiver can then set its time offset according to the transmitted timestamp but has no information about the delay during the transmission. This delay has to be estimated.

Round trip time based synchronisation estimates this delay based on an exchange of two messages, which provides an upper bound on the synchronisation error [41]. In this approach, the needed messages increase linearly with the number of nodes to synchronise.

With reference broadcasts [43], a beacon node sends a broadcast packet. The measured arrival time is used as common time reference in order to synchronise the receivers of the beacon with each other. The synchronisation error is typically smaller than with unidirectional or round-trip synchronisation [40].

For our implementation, we decided to use unidirectional clock synchronisation for two reasons: (1) With the timestamping support of the CC2420, uncertainties in the transmission delay get reasonable small, as shown later in this section. (2) The synchronisation of the network can be done with little effort regarding the amount of messages and the complexity of the implementation. Figure 5-4 shows three phases

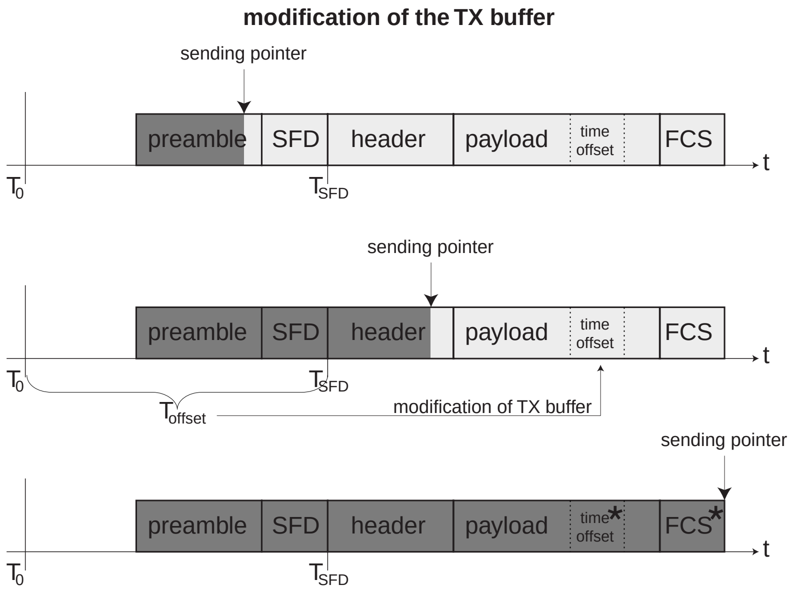


Figure 5-4
Timestamping with the CC2420 on a packet while sending

in the process of timestamping. As soon as the start of frame delimiter (SFD) has been sent, an interrupt is generated which triggers a capture of the current time. Now, time information can be added to the packet being sent. On the receivers end, the arrival of the packet, in particular the SFD, triggers an interrupt again. As we only capture time in the resolution of the ACLK (32.768kHz), the propagation time of the packet is negligible.

The problematic of clock drift in WSNs can be addressed with periodic resynchronisation or estimation of the clock drift [40][42]. We chose the former method, because we already have a periodic message exchange for failure detection and can therefore piggypack synchronisation information on these packets.

5.4.2.1 Synchronisation Performance

In order to estimate the performance of the unidirectional clock synchronisation with the CC2420 MAC layer time stamping feature, we set up a test application with seven nodes. All nodes had a common wake-sleep schedule and were placed in a ring. At the beginning, one node started sending a packet with time information to the next neighbour. The packet was then forwarded from node to node in the ring. The one that had the packet, adjusted its time offset accordingly to the offset information provided from the predecessor. A sniffer node captured all sent packets and logged their SFD time. The gathered information, mapped in figure 5-5, shows the clock drift of each packet relative to the sniffer node's clock. It can be observed, that SFDs are tendentially sent earlier, the more hops involved. We measured a maximal deviation $T_{SyncError,Max}$ per hop of 2 clock ticks which corresponds to $61.035\mu s$.

Applied to our failure detector network, the error of the synchronisation offset at

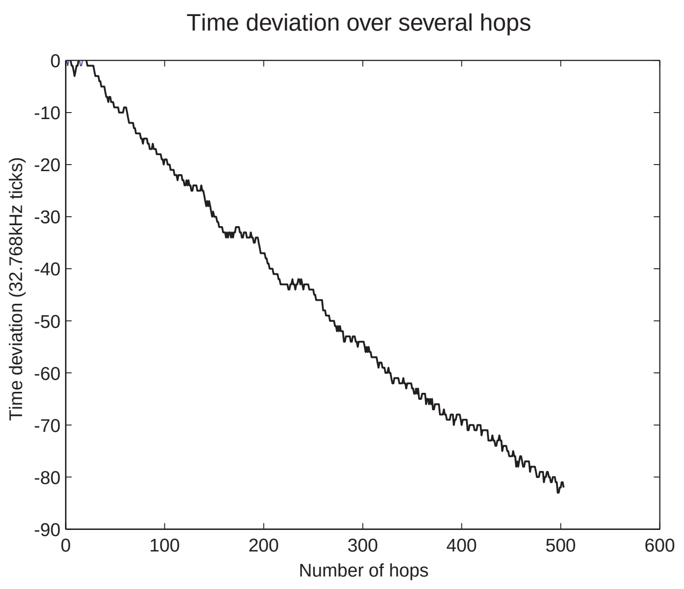


Figure 5-5
Offset deviation with MAC-layer timestamping and unidirectional clock synchronisation

the maximal distance of H hops from the base station amounts is in the range of $\pm H \cdot T_{SyncError,Max}$. Compared to the clock drift of $2ms$ (at a period of $100s$), the maximal synchronisation error of $60\mu s$ is negligible.

5.4.3 Radio Configuration, CC2420 Stack Changes

The chosen MAC layer made it necessary to change the network stack of TinyOS. First of all, we wanted to control the time a packet is sent as precise as possible. This was not possible with the existing implementation. The CCA feature was removed and with this also the need for backoff timers. Furthermore we created an interface that allows a program to load a packet in the radio FIFO and tell the radio explicitly when to switch to the transmitting mode. With this preloading mechanism it is possible to load the frame (header, schedule) of a packet to the FIFO and change some small amount of data (time offset, bitmask) on transmission.

For timing purposes we had to add support for 32-bit timestamps to the stack. This stamping ability is used for saving the time a packet has arrived.

For later extensions, for example for usage in a clustered network, we made use of the address recognition feature of the CC2420 chip. The PAN (personal area network) field in the packet header would give us the opportunity to separate the communication of inter-cluster communication. As soon as a node becomes a member of a cluster, it sets its PAN field to the node id of the cluster head. In order to use this feature, additional commands had to be implemented in the TinyOS network stack.

5.4.4 TDMA Slot Duration

For the implementation of the TDMA scheme, we had to estimate the duration of the slots. As depicted in figure 5-2, a slot is composed of a guard time T_{Guard}, a transmission time $T_{Transmit}$ and a processing time $T_{Process}$. A slot duration can be calculated as follows:

$$T_{Slot} = max(T_{Guard}, T_{Load} + T_{RxTx}) + T_{Transmit} + T_{Process} \tag{5.1}$$

Times concerning the radio chip were measured with the already mentioned experiments in chapter 3. In order to justify the measured values, we calculated some time interval based on values in the datasheet [18]. Those values are listed in table 5-1. The size of a packet can be calulated with following formula:

$$L_{Packet} = L_{Header} + L_{Payload} + L_{CRC} \tag{5.2}$$

where

$$L_{Header} = 11bytes$$

$$L_{CRC} = 2bytes$$

Parameter	**Calculated based on datasheet**	**Measured**
T_{Load}	–	$1.1ms$
$T_{RxTx} + T_{SFD}$	$352\mu s$	$360\mu s$
T_{Tx} (29 Bytes)	$928\mu s$	$1020\mu s$
T_{Tx} (n Bytes)	$n \cdot 32\mu s$	–
$T_{Rx,Stack}$	–	$1.23ms$
$T_{Guard,Report}$	$20ppm$ of T_{Period}	–

Table 5-1: Values for the calculation of slot duration

The guard time T_{Guard} in Equation 5.1 is only necessary for the first reporting wave of each period. After this wave, a resynchronisation happens in the acknowledge wave. This guard time prevents slots from overlapping each other due to clock drift. Otherwise, packet collisions could occur. In the worst case, the clocks of two consecutive nodes drift with maximal deviation in opposite directions. This is illustrated in figure 5-6. T_{Guard} has therefore to be $2 \cdot T_{Drift,WorstCase}$. In the following calculations, we evaluate the slot duration of the first report wave ($T_{Slot,Report,1st}$) and of the report waves of the recovery action ($T_{Slot,Report,Recover}$) individually.

In order to get a good estimation for the processing times, we implemented a first version of the failure detector where these processing times were set very conservatively. With this implementation we measured the time from receiving a packet from the radio stack until it was fully processed by the application. This was done within the program itself, using the local clock. This measurements were logged to the DSN, using the DSN component.

The gathered data gave us a good estimation for the final implementation and showed us potential optimisation possibilities. An example evaluation in Matlab is illustrated in figure 5-7. The histograms shows the application processing time of acknowledge(ACK) packets on several nodes. We recognised relation between TDMA

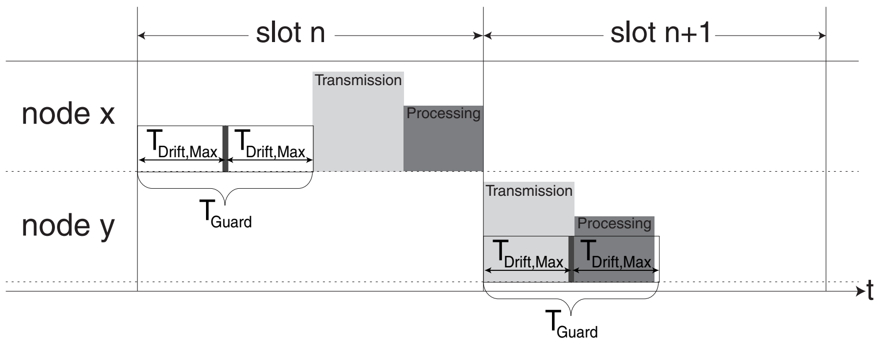


Figure 5-6
Transmission time with worst case clock drift. The guart time T_{Guard} needs to be twice $T_{Drift,WorstCase}$. The local clock of node x is too slow, the clock of node y too fast.

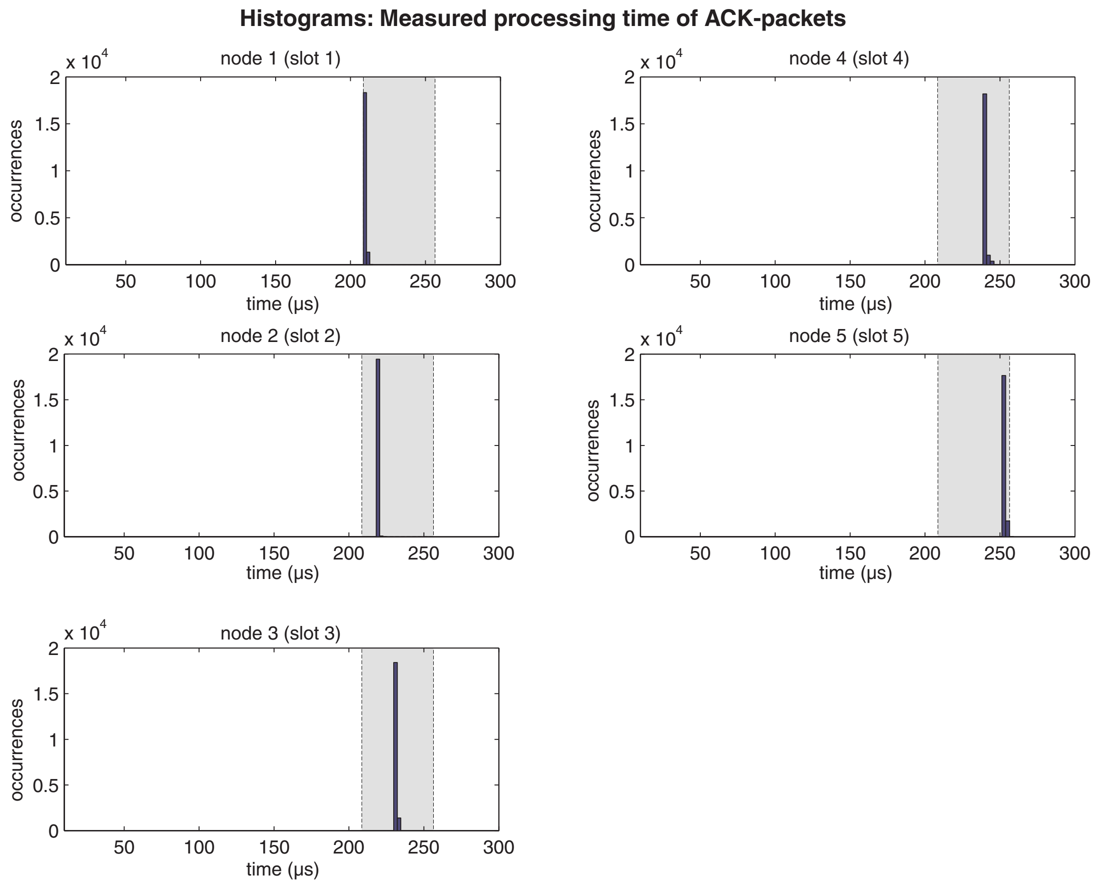


Figure 5-7
Measured application layer processing time of the acknowledge packets. Each plot represents an individual node. In this experiment we could observe, that nodes that are scheduled later have a longer processing time. This insight, supported by the DSN, allowed us to improve the according packet processing steps. Finally, the reporting time depended not anymore on the slot position.

slot position and duration of the application processing time. By examining the code, it turned out, that the reason for this relation was a loop in the timing calculations. Every slot needed an additional iteration. As this code was located in a time critical part, we made an effort to transfer the main calculation component to a less time critical place. In the end, the slot position of a node had no effect on the processing time anymore. The final application processing times are listed in table 5-2.

Parameter	**Measured**
$T_{AppProcess,Report,Max}$	$224\mu s$
$T_{AppProcess,Ack,Max}$	$264\mu s$

Table 5-2: Measured application processing time

If we fill the values from table 5-1 and table 5-2in formula 5.1, we get an estimation for the duration of the reporting and acknowledge slots (T_{Period} is assumed to be $45s$):

$$T_{Guard,Report,1st} = 2ms > T_{Load} + T_{RxTx}$$

$$T_{Transmit,Report} = L_{Packet,Report} \cdot 32\mu s = 896\mu s$$

$$T_{Process,Report} = T_{Rx,Stack} + T_{AppProcess,Report,Max} = 1.23ms + 224\mu s$$

$$T_{Slot,Report,1st} = T_{Guard,Report,1st} + T_{Transmit,Report} + T_{Guard,Report,1st} = 4.35ms$$

$$T_{Guard,Report,1st} = T_{Load} + T_{RxTx} = 1.46ms$$

$$T_{Slot,Report,Recover} = T_{Guard,Report,1st} + T_{Transmit,Report} + T_{Guard,Report,1st} = 3.81ms$$

$$T_{Guard,Ack} = T_{Load} + T_{RxTx} = 1.46ms$$

$$T_{Transmit,Ack} = L_{Packet,Ack} \cdot 32\mu s = 1.216ms$$

$$T_{Process,Ack} = T_{Rx,Stack} + T_{AppProcess,Ack,Max} = 1.23ms + 264\mu s$$

$$T_{Slot,Ack} = T_{Guard,Ack} + T_{Transmit,Ack} + T_{Process,Ack} = 4.17ms$$

The implemented TDMA scheme has still potential for optimisation in terms of low power consumption. For instance, the number of slots could be set adaptive to the number of effectively existing nodes in the cluster. Also possible is a shutdown of the radio module after the information of a node is sent in each wave. The saved power is paid by decreased robustness as alive information cannot be gathered anymore for consecutive wave rounds.

5.4.5 Start-up Phase

The initialisation of wireless sensor networks are rarely discussed in detail in theoretical papers. Our approach was inspired by LEACH [35]: When a node is started, it waits for a random time, listening for an acknowledge wave with synchronisation information. If an expected packet arrives, a joining request is sent if

- the immediate sender of the packet is at least a $(H-1)$-hop neighbour of the sink or the sink itself
- the cluster is not full
- the measured link quality is above a specified threshold

This request is piggypacket on the next report wave. If other requests are made in the same round, they are aggregated in the wave. In an early state of network initialisation, the number of requests is usually high. Therefore we decided to limit the maximal aggregated joining requests per packet. Once a node is registered at the base station, the following acknowledge wave carries its invitation. If a requesting node is not invited, the node switches again to the listening mode for receiving another acknowledge wave. If no sink is present or no joining possible until the random waiting time is over, the node nominates itself as a sink. This behaviour could be used in a clustered network.

5.5 Tests and Analysis

In order to verify our implemented failure detection system, we run some tests. The results are summarised in this section.

5.5.1 Testbed

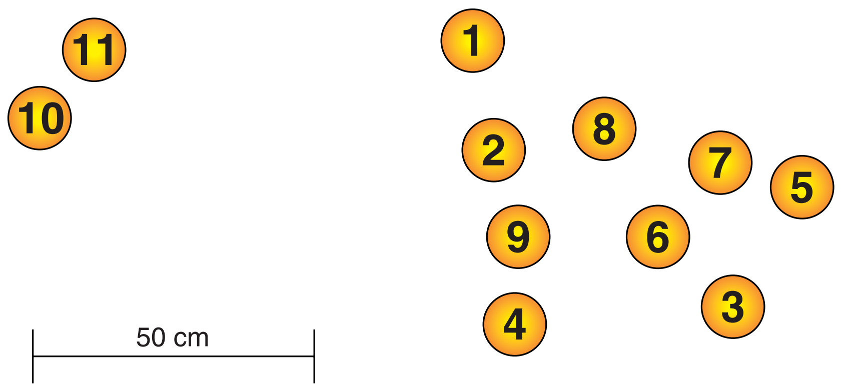
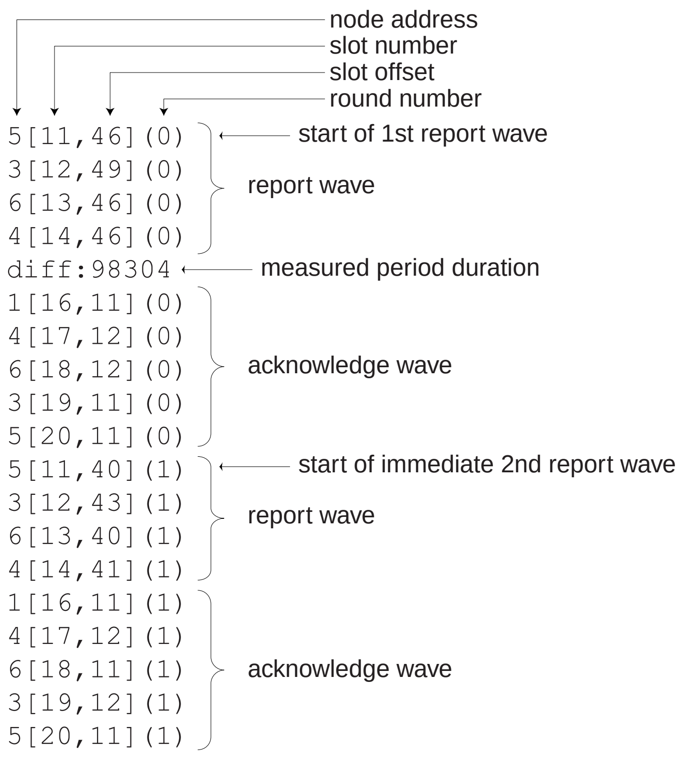


Figure 5-8
Arrangement of noded in the testbed

The following tests were made in a testbed consisting of 11 nodes on a table. Every Tmote had a DSN node attached. To simulate lossy links, we adjusted the transmission power of the failure detector nodes to a minimum. An impression of the setup is given in figure 5-8.

5.5.2 TDMA Schedule Verification

In the implementation phase of the TDMA scheme, we wanted to have a tool to verify the implemented behaviour. This tool should not only be able to display the slot of a transmission but also the exact position in the slot. For this purpose we used one sensor node as a packet monitor. This node was set in a promiscuous listening mode and logged the whole traffic in the cluster. For logging, we used the already described DSN component. In order to get real-time status information, the output of the node had to be made on the integrated USB interface. So we added a software switch to change the logging output between UART0 (DSN-connection) and UART1 (USB). Figure x shows an output example of the sniffer node. This node used its internal clock to measure the arrival times of the packets. From these times, the corresponding slot number and offset were calculated. More flexibility was achieved through adding filter abilities. The final version of the sniffer could filter out packets of a certain type or from a specified sender and display them in a short form, like in figure x, or print out whole packets for detailed debugging. In the printout of the example (Figure 5-9)we see two wave rounds in a cluster with five sensor nodes. The slots are numbered from 0 to 32, which represents the combination of a report wave and an acknowledge wave. Slot 15 is always used for calculations at the base station, so this slot is not logged by the sniffer. Packets of the reporting wave apparently arrive later in a slot. This has to do with the nature of the network initialisation. A report packet could contain a join request of a node. Such a request would need computation time in every forwarding node. For this reason, packet preloading (see section 5.4.3) is not is not applicable here.



```
5[11,46](0)
3[12,49](0)
6[13,46](0)
4[14,46](0)
diff:98304
1[16,11](0)
4[17,12](0)
6[18,12](0)
3[19,11](0)
5[20,11](0)
5[11,40](1)
3[12,43](1)
6[13,40](1)
4[14,41](1)
1[16,11](1)
4[17,12](1)
6[18,11](1)
3[19,12](1)
5[20,11](1)
```

Figure 5-9
Output of the packet sniffer in a network of 5 nodes. In this case, a normal round and one recover round is logged. For slot number and offset measuring, the first packet from the base station is used as time base. Packets from the reporting wave are sent later in a slot than packets of the acknowledge wave. This is due to the preloading feature used for the later wave.

5.5.3 Reporting Time Measurement

We measured the reporting time of our implementation in an experimental setup. In order to eliminate environmental impact on the linkquality, we run the testbed at maximal transmission power. Particular nodes were then selectively shut down by using the target command feature of the DSN. A Java program was used to inject those *shutdown-commands* at a random time. Three runs with different period time T_{Period} were made. We set the timeout for an entry in the node table to $90s$.

The results, shown in figure 5-10, look as could be expected. As no retransmissions were needed, missing nodes could be reported within a period. The shorter the period is, the faster the reaction of the system. Some measured reporting times were slightly (maximal $2s$) higher than $90s$. This can be traced back to the accuracy of the logging timestamps of the DSN.

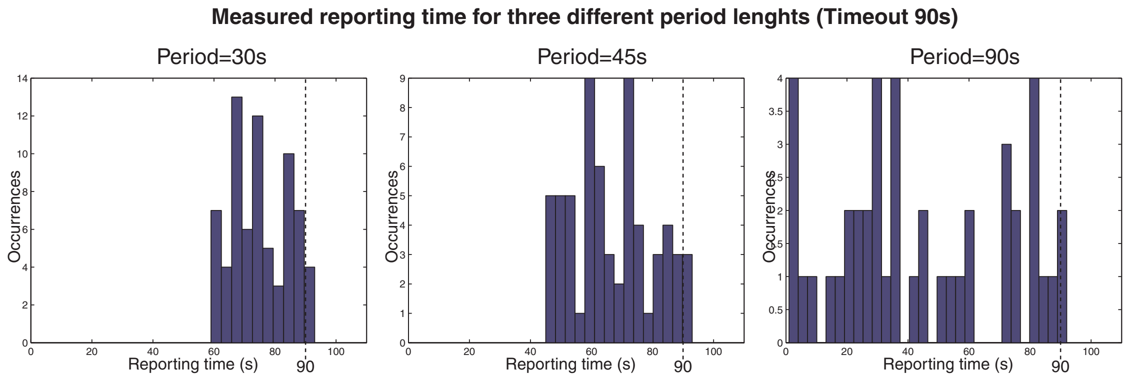


Figure 5-10
Histogram of the measured reporting times with different periods T_{Period}

5.5.4 Power Consumption Estimation

5.5.4.1 Measuring Rounds per Period

For power consumption estimation, we used our model in chapter 3. First, we have to calculate the mean current over a period. This value is not always constant because of retransmissions. In order to get example data from our failure detector, we added round logging support. Each node reported the number of wave rounds used in each period. The samples were then gathered with the attached DSN. The testbed run with the parameters listed in table 5-3 and the calculated duration of the slots from section 5.4.4. We run a total of three test cycles, each time with a different transmission power set. As the nodes in the testbed had little distance to each other we configured the radio module with the three lowest possible values for transmission power.

The resulting data from these runs is shown in figures 5-11 to 5-13. We grouped the curves according to the hop count of the nodes. For better readability, the curves were filtered by a sliding average window of 50 samples.

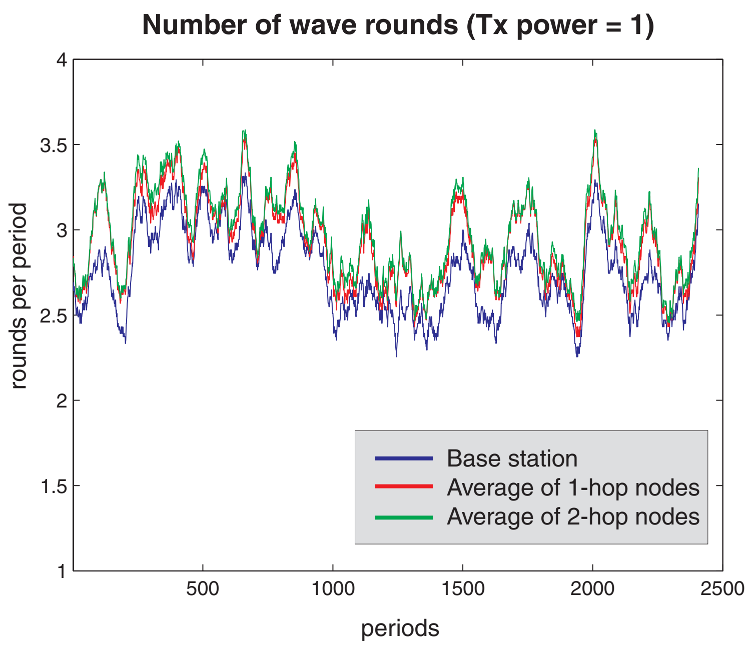


Figure 5-11
Plot of the round count experiment at a transmission power parameter of 1

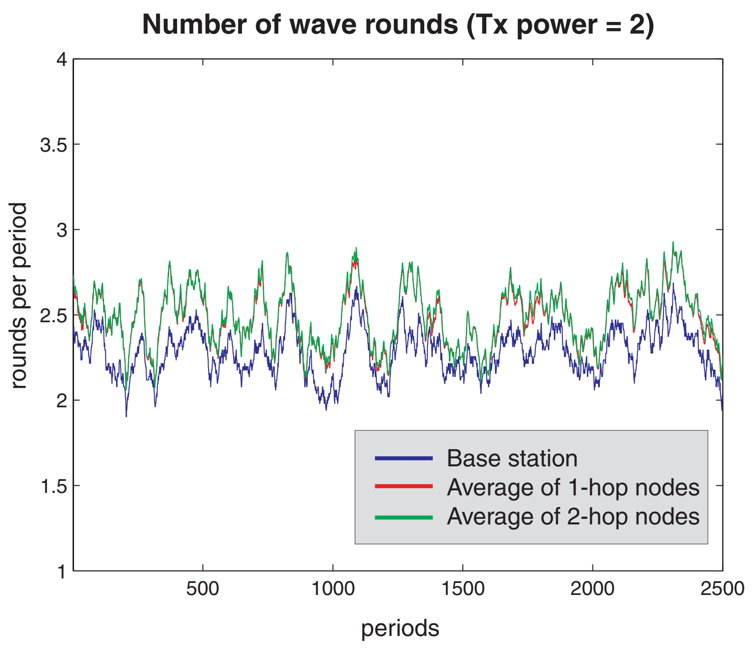


Figure 5-12
Plot of the round count experiment at a transmission power parameter of 2

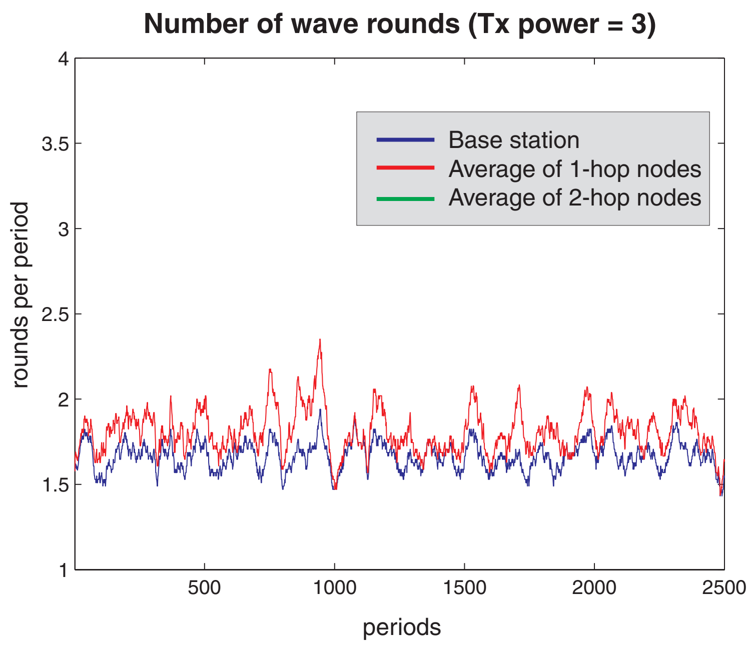


Figure 5-13
Plot of the round count experiment at a transmission power parameter of 3

Parameter	**Value**
T_period	$45s$
Maximal retries	3
Transmission power	$\{1, 2, 3\}$

Table 5-3: Parameters for the round count experiment.

Nodes with a higher hop count seem to run more additional rounds than those with lower distance to the sink. This can be explained with the fact, that a wave has to be forwarded from the inner nodes to the outer ones. A message for an outer node has to be transmitted over twice as much links, so the probability of a link failure increases. Comparing the three test runs with each other, we can observe, that transmission power increases the robustness of the network. The higher the power, the lesser rounds are counted per period. In the last run (Figure 5-11), the radio range is so high, that the initialisation procedure of the network does not anymore recognise 2-hop nodes.

5.5.4.2 Lifetime Estimation

With the measured data in the previous section, we could estimate the lifetime for each setup. As the energy consumption is individual for each node, we define the lifetime of the network as the time, when the first node's energy is used up. Additionally, we were also interested in the theoretical minimum and maximum lifetime, limited by the capacity of the battery. The following calculations characterise the network profile for our estimation. Time intervals had to be calculated for the scenarios, where one or more wave rounds happened in a period.

The time used for transmission in a round (T_{Tx}) and the idle listening time ($T_{idle/rx,1st}$, $T_{idle/rx,recover}$) were calculated based on values derived in section 5.4.4:

$$T_{Tx} = 2 \cdot T_{RxTx} + T_{Tx,Report} + T_{Tx,Ack} = 2.83ms$$

$$T_{idle/rx_1st} = N \cdot (T_{Slot,Report,1st} + T_{Slot,Ack}) - T_{Tx} = 133.49ms$$

$$T_{idle/rx_recover} = N \cdot (T_{Slot,Report,Recover} + T_{Slot,Ack}) - T_{Tx} = 124.85ms$$

From this values, we calculated the average current in each case, according to our model in section 3.3.3. These values are shown in table 5-4. In Table 5-5 we present

Rounds per period (n)	**Average current (I_{Avg})**
1	$98.96\mu A$
2	$157.67\mu A$
3	$215.61\mu A$
4	$274.31\mu A$

Table 5-4: Average currents for n rounds per period

the theoretical lifetime of our implementation if the node were energised by a typical NiMH cell. We assumed a battery capacity of 2300mAh.

Setup	**Average current (I_{Avg})**	**Calculated lifetime in years**
Max. current	$274.31\mu A$	0.96
Min. current	$98.96\mu A$	2.65
Tx power 1	$225.1\mu A$	1.17
Tx power 2	$187.4\mu A$	1.40
Tx power 3	$146.2\mu A$	1.80

Table 5-5: Calculated network lifetime with a 2300mAh battery. Theoretical bounds and realistic values, based on round measurements.

6

Conclusion and Outlook

Development of embedded distributed systems is a very hard and time consuming task. During our implementation of the network monitoring application (and also several test applications), a lot of time has been invested in debugging. In this area, support for implementation and deploying is substantial in order to get satisfying results in a reasonable time. As a premier to the Tmote Sky platform, we could use the JAWS deployment support network for gathering information from the whole target network.

With TinyOS 2.0, we used a wireless sensor network software framework that is still in development. Although we could not use the end product, we think TinyOS 2.x has great potential to continue the successful history of its predecessor TinyOS 1.x. The community around TinyOS and Tmote Sky is very active, which promise further improvements in software architecture.

6.1 Contributions

We implemented a package for Tmote Sky that enables the support with the JAWS deployment support network also to other developers on this hardware platform. This package consists of three elements:

1. The *hardware connection* between Tmote Sky and a DSN node (BTnode rev3).
2. The *NesC component* that enables TinyOS 2.0 applications to log messages to the DSN, receive target commands and set the network identifier of each Tmote Sky node.
3. A TI-BSL protocol compliant *bootloader* for programming the microcontroller MSP430 over its non-standard serial interfaces UART0.

This DSN package had been used for implementing a heartbeat style network monitoring system. The features of the application are listed below:

- A given *reporting time limit* can be guaranteed with very high probability.

- With support of redundant information flow and acknowledged messaging, the monitoring facility is not affected by temporary link failures or channel failures.
- With a TDMA scheme, the application stays mostly in a sleep mode and is therefore applicable for battery powered long term deployment.

In order to profile the implemented system, we made following measurements:

- Time and current on the Tmote Sky platform, focused on the sending and receiving processes.
- Network wide data gathering concerning retransmissions and several time aspects (reporting, processing, TDMA schedule).

The observed data can be used as base for further implementations on the Tmote Sky platform. With the parameters for our lifetime estimation model, other Tmote WSN applications can be characterised.

6.2 Further Work

Our DSN component for TinyOS worked well with the version from the development branch. The final release is expected to have changes that will make adaptions necessary to our component. Changes will mostly affect lower level UART interfaces.

The implementation of a monitoring system has still potential for optimisation an enhancement. We see the biggest opportunities in following points:

Low Power Altough already trimmed, further energy savings could be achieved with a smart wakeup-sleep scheme during the active phase (the wave round) of the TDMA schedule.

Redundant sink If the base station is inteneded to forward detected errors, it gets a single point of failure in the forwarding chain. This could be avoided with a second sink.

Larger networks For large networks, the used mechanism gets inefficient. A clustering approach could provide the needed scalability.

Fast event forwarding This ability is needed for security related reporting systems like example fire sensor networks.

A

Aufgabenstellung

Einleitung

Ein drahtloses Sensor Netzwerk (WSN—Wireless Sensor Network) besteht aus einer Vielzahl von kleinen resourcenbeschränkten Knoten welche mit Funkmodul und Sensoren bestückt sind. Diese werden in der Umwelt (z.B. in einem Haus) verteilt und erstellen möglichst autonom ein Netzwerk. Ein solches Netz ermöglicht den Knoten Sensor-Messungen auszutauschen und diese Daten gemeinsam zu verarbeiten. Nach einer Vision von Stankovic et al. [?] soll dies die 'nahtlose Integration von Rechner mit der Umwelt mit Hilfe von Sensoren und Aktoren ermöglichen'.

Ein Anwendungsszenario für solche WSNs ist die Sammlung der Sensordaten in einem designierten Knoten (Senke/Zentrale), welcher anhand der erhaltenen Information Entscheidungen treffen muss. Vielfach sind die Knoten physikalisch weiträumig verteilt, was der Mehrheit der Knoten verunmöglicht direkt mit der Senke zu kommunizieren. Stattdessen müssen die Daten über mehrere Knoten/Hops zur Senke geschickt werden (siehe Abbildung 1). In verschiedenen Projekten [?, ?] konnte in den vergangenen Jahren Erfahrungen mit solchen Multihop Sensor-Netzwerken gemacht werden. Dabei wurde festgestellt, dass (aus Gründen wie temporärer Linkausfall und Interferenz) ein nicht zu vernachlässigbarer Teil der verschickten Daten nicht bei der Senke ankommt. D.h. die Daten werden mit "best effort" verschickt und es gibt keinerlei Garantien, dass die Daten auch wirklich bei der Senke ankommen.

Ein konkretes Beispiel eines Sensor Netzwerkes ist die Sammlung von Feuermelder-Daten bei einer Zentrale. Heutzutage sind die Feuermelder entweder fest verdrahtet oder in direktem Funkkontakt mit Basisstationen,

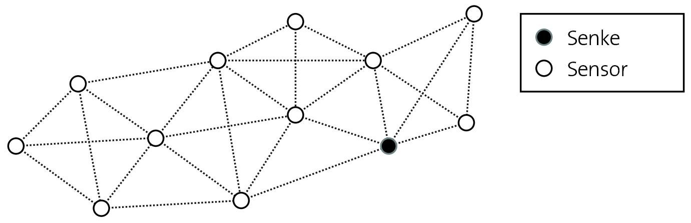


Abbildung 1: Schema eines Sensor Netzwerkes. Nicht alle Knoten haben eine direkte Verbindung zur Zentrale (Senke) und müssen die Daten über mehrere Hops gesendet werden.

welche ihrerseits per Kabel mit der Zentrale verbunden sind. In beiden Fällen ist es somit nötig Kabel im Gebäude zu verlegen. Das ist einerseits kostenintensiv und andererseits aus bautechnischen Gründen nicht immer möglich (z.B. denkmalgeschützte Kirche). Um die Feuermelder gänzlich ohne Verkabelung zu installieren, ist es nötig, dass die Feuermelder untereinander ein Netzwerk aufbauen welches Feuermeldungen über mehrere Hops verschicken kann.

In dieser Arbeit soll ein Demonstrator eines solchen drahtlosen Feuermelde-Netzwerkes (FSN—Fire-Sensor Network) aufgebaut werden. In einem FSN dürfen die oben erwähnten Datenverluste keinesfalls vorkommen—ein Feuer das detektiert wird, muss auch verlässlich bei der Zentrale gemeldet werden. In einem FSN ist man mit folgenden Anforderungen konfrontiert:

1. Die Zustellung der Feuermeldung muss zuverlässig innerhalb einer bestimmten Zeitlimite (∼10 Sekunden) erfolgen. D.h. es gibt Echtzeitanforderungen.
2. Die Zentrale kann bei einer Feuermeldung einen Knoten veranlassen einen Aktuator (akustisches Signal) zu aktivieren. Dies muss innerhalb einer bestimmten Zeitlimite geschehen (∼20 Sekunden nach der Feuerdetektion).
3. Knoten können ausfallen. Das Netzwerk muss solche Ausfälle erkennen und ebenfalls in einer bestimmten Zeitlimite (∼100 Sekunden) der Zentrale melden.
4. Einzelne Links können temporär ausfallen (z.B. eine Türe wird geschlossen oder ein Schrank wird verschoben).
5. Eine bestimmte Frequenz kann temporär ausfallen (z.B. ein Garagentüreöffner blockiert für einige Sekunden eine bestimmte Frequenz), was eine zuverlässige Kommunikation verunmöglicht. In einem solchen Fall muss auf eine alternative Frequenz ausgewichen werden.
6. Die Batterie-betriebenen Feuermelder haben nur sehr begrenzt Energie zur Verfügung. Es ist deshalb notwendig den Knoten mehrheitlich auszuschalten ('Duty Cycle' ∼1/1000) und somit oft nicht zur Verfügung stehen um eine Feuermeldung weiter zu leiten.

Der Demonstrator hat die ersten fünf Punkte zuverlässig zu erfüllen. Dies mit der Vorgabe, den Energieverbrauch zu optimieren. Dabei muss berücksichtigt werden, dass die Lebensdauer des ganzen Netzwerkes ausschlaggebend ist—massgebend ist somit der erste Knoten dem die Energie ausgeht.

Beim Aufbau eines WSNs ist man mit der Problematik konfrontiert, dass man nicht genau weiss was in den einzelnen Knoten geschieht. Es wäre zwar prinzipiell möglich zusätzliche Information über das WSN verschicken, jedoch wird dies höchstwahrscheinlich das Verhalten des Netzwerkes verändern. Zudem kann es auch gut sein, dass die Kommunikation noch nicht zuverlässig funktioniert, und es deshalb gar nicht erst möglich ist, Zugang zum Netzwerk und somit den Knoten zu erhalten. In dieser Arbeit soll für das Fehlersuchen und Softwareupdaten eine so genanntes 'Deployment Support Network' (DSN) benutzt werden. Ein DSN ist ein kabelloses Sekundärnetzwerk und ermöglicht die Entwicklung, das Testen, und die Validierung von Sensor-Applikationen [?]. Dazu werden die WSN/DSN Knotenpaare gebildet welche mit einem kurzen Kabel verbunden sind. Die DSN Knoten bauen eigenständig ein drahtloses Netzwerk auf und ermöglichen somit, wie in Abbildung 2 dargestellt, einen einfach Zugang zu den angehängten WSN Knoten.

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest [?]. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze, Systeme, Software und Fastprototyping vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie auch nach relevanten neueren Publikationen. Vergleichen Sie bestehende Demonstratoren anderer Universitäten. Prüfen Sie welche Ideen/Konzepte Sie aus diesen Lösungen verwenden können.

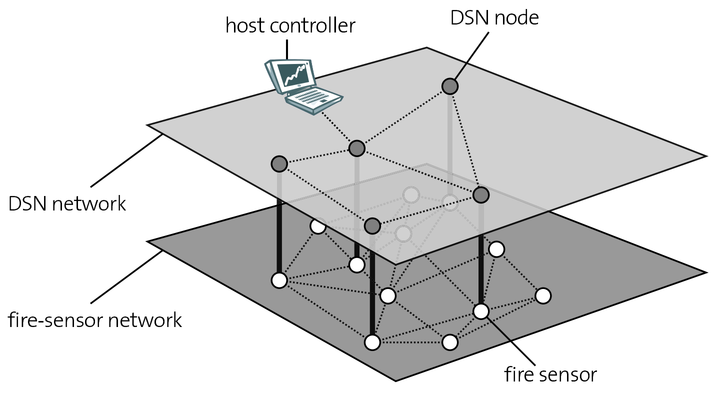


Abbildung 2: Die Knoten des Feuermeldenetzwerkes (FSN) sind per Kabel mit den Knoten des 'Deployment Support Network' (DSN) verbunden. Dieses Sekundärnetz erlaubt einen einfachen zugriff auf die Knoten angeschlossenen Knoten des FSN. Insbesondere vereinfacht dies die Fehlersuche sowie das Update der FSN-Software.

3. Die FSN Knoten sollen auf der Tmotes Sky Platform [?] implementiert werden. Arbeiten Sie sich in die Softwareentwicklungsumgebung der Knoten und dem Bertriebssystem TinyOS [?] ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (Versionskontrolle, Bugtracker, online Dokumentation, Mailinglisten, Application Notes, Beispielapplikationen).
4. Nehmen Sie das JAWS Deployment-Support Network [?] auf einigen Knoten in Betrieb und testen Sie dieses auf Zuverlässigkeit und Leistung.
5. Bauen Sie mit Hilfe des DSNs einen Demonstrator eines Feuermeldenetzwerkes auf.
   (a) Konzeptentwurf für die Protokolle und Applikation
   (b) Strategie/Spezifikation der Implementation
   (c) Implementation des Demonstrators
6. Testen Sie das Feuermeldenetzwerkes mit Hilfe des DSNs:
   (a) Mit Hilfe des DSN wird bei einem Knoten des WSNs ein Feuer simuliert. Kommt die Feuermeldung rechtzeitig bei der Zentrale an?
   (b) Das DSN teilt einem Knoten mit, auf Stumm zu schalten. Das Netzwerk hat diesen Knoten in der erforderlichen Zeit zu melden. Ist die Zustellung einer Feuermeldung immer noch zuverlässig wenn ein einzelner Knoten stumm ist?
   (c) Mit Hilfe des DSN sollen Kanal und Linkfehler simuliert werden. Funktioniert die Zustellung der Feuermeldung immer noch zuverlässig?
7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Masterarbeit

Allgemeines

- Der Verlauf des Projektes Masterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.

- Sie verfügen über einen PC mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Masterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern.
- Sie führen ein Researchtagebuch in welchem sie die Fortschritte täglich protokollieren.

Abgabe

- Geben Sie vier unterschriebene Exemplare des Berichtes, das Researchtagebuch sowie alle relevanten Source-, Object und Konfigurationsfiles bis spätestens am 15. Oktober 2006 dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Bibliography

[1] K. Langendoen and G. Halkes, *Energy-Efficient Medium Access Control,*
in The Embedded Systems Handbook, CRC press, August 2005

[2] J. Polastre, J. Hill and D. Culler, *Versatile low power media access for wireless sensor networks,*
SenSys04, 2004

[3] W. Ye, J. Heidemann and D. Estrin, *An Energy-efficient MAC Protocol for Wireless Sensor Networks,*
21st Conference of the IEEE Computer and Communications Societies (INFOCOM), 2002

[4] A. El-Hoiydi and J.-D. Decotignie, *WiseMAC: An Ultra Low Power MAC Protocol for Multi-hop Wireless Sensor Networks,*
First Int. Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS 2004), Lecture Notes in Computer Science, LNCS 3121, 2004

[5] K. Arisha, M. Youssef and M. Younis, *Energy-Aware TDMA-Based MAC for Sensor Networks,*
IEEE Workshop on Integrated Management of Power Aware Communications, Computing and NeTworking (IMPACCT 2002), 2002

[6] G. Pei and C. Chien, *Low Power TDMA in Large Wireless Sensor Networks,*
MILCOM01, 2001

[7] J. Li and G. Lazarou, *A bit-map-assisted energy-efficient MAC scheme for wireless sensor networks,*
IPSN04, 2004

[8] S. Kulkarni and M. Arumugam, *SS-TDMA: A Self-Stabilizing MAC for Sensor Networks,*
Sensor Network Operations, 2006

[9] L.F.W. van Hoesel, T. Nieberg , H.J. Kip , P.J.M. Havinga, *Advantages of a TDMA based, energy-efficient, self-organizing MAC protocol for WSNs*

[10] L. van Hoesel and P. Havinga, *A Lightweight Medium Access Protocol (LMAC) for Wireless Sensor Networks,*
INSS04, 2004

[11] S.-C. Wang and S.-Y. Kuo, *Communication strategies for heartbeat-style failure detectors in ad hoc networks,*
in Proceedings of the International Conference on Dependable Systems and Networks (DSN-2003), (San Francisco, CA), pp. 361370, June 2003.

[12] Ann T. Tai, Kam S. Tso, William H. Sanders, *Cluster-Based Failure Detection Service for Large-Scale Ad Hoc Wireless Network Applications,*
in Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN04)

[13] *TinyOS 2.x,*
http://www.tinyos.net/tinyos-2.x/doc/

[14] *MoteIV ,*
http://www.moteiv.com/

[15] J. Polastre, R. Szewczyk, D. Culler, *Telos: enabling ultra-low power wireless research,*
in Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN '05),Los Angeles, California, 2005

[16] *Tmote Sky datasheet,*
Moteiv Corporation, 2006

[17] *TI MSP430x1xx User's Guide,*
Texas Instruments Incorporated, 2006

[18] *CC2420 Datasheet, 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver,*
Chipcon AS

[19] *IEEE 802.15 TG4,*
http://www.ieee802.org/15/pub/TG4.html

[20] P. Levis, D. Gay, D. Culler, V. Handziski, J. Hauer, C. Sharp, B. Greenstein, J. Hui, J. Polastre, R. Szewczyk, K. Klues, G. Tolle, L. Nachman, A. Wolisz, *T2 - What the Second Generation Holds,*
http://csl.stanford.edu/ pal/talks/berkeley-seminar-05.pdf#search=%22tinyos%202.0%22

[21] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz and D. Culler, *TEP 2: Hardware Abstraction Architecture,*
http://www.tinyos.net/

[22] K. Klues, P. Levis, D. Gay, D. Culler, V. Handziski, *TEP 108: Resource Arbitration,*
http://www.tinyos.net/

[23] R. Szewczyk, P. Levis, M. Turon, L. Nachman, P. Buonadonna, V. Handziski, *TEP: Microcontroller Power Management,*
http://www.tinyos.net/

[24] J. W. Hui, D. Culler, *The dynamic behavior of a data dissemination protocol for network programming at scale*, in Proceedings of the 2nd international conference on Embedded networked sensor systems, November 2004

[25] K.-Y. Jang, M. Vieira, S. Rangwala, O. Gnawali, R. Govindan, *Tutornet at University of Southern California*,
http://enl.usc.edu/projects/tutornet/index.html

[26] S.-F. Li, V. Handziski, A. Köpke, M. Kubisch, A. Wolisz, *A Wireless Sensor Network Testbed Supporting Controlled In-building Experiments*,
in Proc. of 12. Sensor Kongress, Nrnberg, Germany, May 2005

[27] *Stargate - XScale Network Interface and Single Board Computer*,
Crossbow Technology Inc., http://www.xbow.com/

[28] *sMote, Omega* ,
http://www.millennium.berkeley.edu/sensornets/

[29] G. Werner-Allen, P. Swieskowski, M. Welsh, *MoteLab: A Wireless Sensor Network Testbed*,
in Proceedings of the 4th international symposium on Information processing in sensor networks IPSN '05 , April 2005

[30] M. Dyer, J. Beutel, L. Meier, *Deployment Support for Wireless Sensor Networks*,
KuVS Fachgesprch Sensornetzwerke ETH Zurich, 2005

[31] *BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks*,
http://www.btnode.ethz.ch/

[32] *TinyNode* ,
Shockfish SA, http://www.tinynode.com/

[33] S. Schauer, *Features of the MSP430 Bootstrap Loader*,
Application Report, Texas Instruments, February 2006

[34] D. Hobi and L. Winterhalter, *Large-scale Bluetooth Sensor-Network Demonstrator*,
Master Thesis, ETH Zurich, 2005

[35] W. Heinzelman, A. Chandrakasan, H. Balakrishnan, *Energy-efficient Communication Protocols for Wireless Microsensor Networks*,
International Conference on System Sciences, January 2000

[36] O. Couach, E. Bou-Zeid, H. Huwald, M. Parlange, *Une tude de terrain sur le glacier de la Plaine Morte*,
Bulletin technique de la Suisse Romande "Tracs" n9, mai 2006, p. 15-16.

[37] K. Martinez, R. Ong, J. K. Hart, and J. Stefanov, *GLACSWEB: A Sensor Web for Glaciers. In Adjunct Proc. EWSN*,
Berlin, Germany, January 2004.

[38] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, M. Welsh, *Monitoring Volcanic Eruptions with a Wireless Sensor Network*,
in Proc. European Workshop on Sensor Networks (EWSN’05), January 2005

[39] *Habitat Monitoring on Great Duck Island*,
http://www.greatduckisland.net/

[40] K. Römer, P. Blum, L. Meier, *Time Synchronization and Calibration in Wireless Sensor Networks*,
In: I. Stojmenovic (Ed.), Handbook of Sensor Networks: Algorithms and Architectures, pp. 199-237, Wiley and Sons, October 2005

[41] F. Cristian, *Probabilistic clock synchronization*,
Journal of Distributed Computing, 3:146-158, 1989.

[42] F. M. Gardner, *Phaselock Techniques*,
Wiley, 1979.

[43] J. Elson, L. Girod, D. Estrin, *Fine-grained network time synchronization using reference broadcasts*,
In Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), December 2002

[44] *Datasheet CMR200T, CMR250T*,
http://www.citizencrystal.com/images/pdf/ks-cmr.pdf