DISS. ETH NO. 18959

Ultra-Low Power Sensor Networks: Development Tools, Design, and Implementation

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

NICOLAS BURRI

MSc ETH, ETH Zürich

born 02.05.1978

citizen of
Basel (BS)

accepted on the recommendation of

Prof. Roger Wattenhofer, examiner
Prof. Jochen Schiller, co-examiner

2010

Abstract

Wireless Sensor Networks (WSN) are a powerful tool for the surveillance of environmental conditions, natural habitats, or industrial machinery. All these applications require a long, independent operation of the network without human intervention. The lifetime of a network is limited by the restricted energy resources of the individual deployed sensor nodes. To maximize network lifetime it is therefore essential to minimize the power consumption of all nodes. To achieve energy-efficiency sensor nodes have to be kept in a sleep state in which they are neither able to execute computational tasks nor to communicate among themselves.

In this thesis we consider the design and implementation of energy-efficient communication protocols and applications for sensor networks. The thesis is organized in two orthogonal parts. The first part is dedicated to the Dozer project. Dozer is a communication system designed and optimized for applications in the domain of environmental monitoring. At a sample rate of one reading per two minutes, sensor nodes achieve a lifetime of up to ten years. This makes Dozer the most energy efficient system for applications with continuous, low data rates.

The second part of this thesis discusses development tools for sensor networks. With an application providing control and monitoring functionality and an integrated development environment on Eclipse-basis we provide TinyOS developers with tools to design, implement, and test their protocols more efficiently.

Zusammenfassung

Drahtlose Sensornetzwerke sind ein mächtiges Werkzeug zur Überwachung von Umweltbedingungen, Lebensräumen und industriellen Anlagen. Bei all diesen Anwendungsszenarien ist ein langer, selbständiger Betrieb des Netzes ohne menschliche Unterstützung erforderlich. Die Lebensdauer eines Sensornetzes wird durch die begrenzten Energieressourcen der einzelnen Sensorknoten bestimmt. Um die Lebensdauer zu maximieren, ist es daher essentiell, den Energieverbrauch der Knoten zu minimieren. Um Energie zu sparen, müssen die Sensorknoten den Grossteil der Zeit in einem Schlafmodus gehalten werden, in dem sie weder Berechnungen durchführen noch kommunizieren können.

In dieser Arbeit beschäftigen wir uns mit der Entwicklung energieeffizienter Kommunikationsprotokolle und Anwendungen für Sensornetze. Die Arbeit ist in zwei zu einander orthogonale Teile gegliedert. Der erste Teil ist dem Dozer Projekt gewidmet. Dozer ist ein Kommunikationssystem für Sensornetze, das für Anwendungen im Bereich der Umweltüberwachung optimiert wurde. Bei einer Datenerfassungsrate von einer Messung alle zwei Minuten, erreichen Sensorknoten mit Dozer eine Laufzeit von bis zu 10 Jahren. Damit ist Dozer das zurzeit energieeffizienteste System für Anwendungen mit kontinuierlicher, niedriger Datenrate.

Der zweite Teil der Arbeit behandelt Entwicklungswerkzeuge für Sensornetze. Mit einer Applikation zur Überwachung und Steuerung von Sensornetzen sowie einer integrierten Entwicklungsumgebung auf Eclipse-Basis, bieten wir TinyOS Entwicklern eine Möglichkeit ihre Protokolle effizienter zu implementieren und zu testen.

Acknowledgements

A dissertation is never the work of one person and therefore I would like to thank all the people who have helped and supported me during my time as a PhD student. First of all, I would like to thank my advisor Roger Wattenhofer. I am very grateful for the opportunity you gave me to work with you and for your constant efforts to unite the theoretical and systems aspects of distributed computing. I would also like to thank my co-examiner Jochen Schiller for serving on my committee board despite his very tight schedule.

Furthermore, my thanks go to my colleagues at DISCO (nee DCG). I would especially like to thank Pascal von Rickenbach, my longtime office mate and second parent of Dozer. The many night shifts we spent in the office would not have been as easy without you and your special sense of humor. Also I would like to thank Remo Meier and Pascal for showing me a way in the post-DCG life by founding the startup StreamForge together with me. In this context I would also like to thank Benjamin Sigg for joining us and for his patience with me when I kill the database with bad entries.

My thanks also go to Raphael Eidenbenz for being a loyal co-coffee drinker, Roland Flury for trying to make TinyOS a better system, Olga Goussevskaia for bearing with us when we spoke Swiss German, Michael Kuhn for not finishing his PhD before I did, Christoph Lenzen for providing a mathematicians point of view to many problems, Thomas Locher for appreciating Matt Groening, Johannes Schneider for organizing the poker nights, Philipp Sommer for taking over the sensor network development at DISCO, Yvonne Anne Pignolet for not being resentful, Stefan Schmid for proving that the world is not such a dangerous place, and Thomas Moscibroda for being almost as sarcastic as Pascal. I would also like to thank the next generation of PhDs, Stephan Holzer, Barbara Keller, Tobias Langner, Jasmin Smula, and Samuel Welten for maintaining the traditional addiction of the group to the Toeggelikasten and for providing the exceptions to this rule (for the sake of privacy I will not reveal who belongs to which group). Last but not least I would like to thank the "first generation", Keno Albrecht, Fabian Kuhn, Regina O'Dell, and Aaron Zollinger for getting me started at DCG and for being da cool gang. My special thanks go to Keno, my first office mate, for sharing his extracurricular knowledge with me and for being a worthy opponent in any competition with the staple gun.

I would also like to thank the sensor network guys at TEC, especially Jan Beutel, Roman Lim, and Mustafa Yuecel for the many interesting discussions and their support whenever a hardware related matter went beyond my understanding.

I also had the pleasure to work with many talented students while supervising their diploma and master theses and I would like to thank all of them for this experience. My special thanks go to Roland Schuler and Benjamin Sigg for their exceptional work on the Yeti project and to Otmar Caduff for his excellent work on the sensor network monitoring tool.

Finally, my greatest thanks go to my family for their constant support during all the ups and downs of the last years. I will forever be indebted and grateful to my parents Marlene and Roland for their love and encouragement I received during all my years at ETH. I would also like to thank my brother Alain, his wife Sabrina, and their baby boys Manuel and Adrian for sharing with me what it means to get kids.

Contents

Introduction **7**

I The Dozer Project **11**

1 Introduction **13**
1.1 Origin 13
1.2 Data Gathering 15
1.3 Design Philosophy 16

2 Dozer System **19**
2.1 Dozer Overview 19
2.2 Dozer Implementation 21
2.2.1 Tree Maintenance 21
2.2.2 Scheduler 23
2.2.3 Data Administration 25
2.2.4 Command Management 27

3 Evaluation **29**
3.1 Hardware and Operation System 29
3.2 Small Scale Experiments 31
3.3 Office Floor Experiment 33
3.3.1 Setting and Protocol Parameters 33
3.3.2 Tree Topology 36
3.3.3 Energy Consumption 37

4 Lessons Learned **41**
4.1 Testing Cannot be Outsourced and Requires Adequate Testbeds 41
4.2 Meaningful Tests Take Time 42
4.3 Know Your OS and Hardware 42

4.4 Debug Information 43
4.5 Offline Phase 44
4.6 Self-Repair 45
4.7 Software Updates 46

5 Dozer Revisited 49
5.1 Time Synchronization 49
5.1.1 Fundamentals 50
5.1.2 Clock Synchronization 50
5.1.3 Time Synchronization 51
5.2 Clock Drift Compensation in Dozer 52
5.3 Experimental Evaluation 55
5.3.1 Office Floor Experiment 55
5.3.2 Multiple Sinks 58
5.3.3 Outdoor Experiments 59

6 Dozer in the Wild 65
6.1 About PermaSense 65
6.2 Dozer to PermaDozer 66
6.3 Performance 69
6.4 Conclusions 71

7 Related Work 73
7.1 Comparison 75
7.1.1 LPL and Twinkle 75
7.1.2 KOALA 77
7.1.3 TSMP 79
7.1.4 IP is Dead, Long Live IP for Wireless Sensor Networks 80
7.2 More Related Work 82

8 Conclusions and Outlook 85

II Developement Support for Wireless Networks 87

9 Introduction 89

10 Simulation of Ad Hoc Networks 91
10.1 SANS 92
10.1.1 Design Goals 93
10.1.2 Overview 93
10.1.3 Simulation of Physical and Data Link Layers 95
10.1.4 Internal Network Simulation 97

10.1.5 Related Work . . . 98
10.1.6 Concluding Remarks . . . 99

11 Monitoring Sensor Networks **101**
11.1 Overview . . . 102
11.1.1 Workstation Application . . . 102
11.1.2 Sensor Network . . . 103
11.2 Remote Procedure Calls . . . 104
11.3 Logging . . . 104
11.3.1 Design Goals . . . 105
11.3.2 Logger . . . 105
11.3.3 Remote Log Reader . . . 106
11.4 Topology Control . . . 107
11.4.1 Physical Neighborhood . . . 107
11.4.2 Virtual Overlay . . . 107
11.5 Concluding Remarks and Outlook . . . 108

12 YETI **111**
12.1 Introduction . . . 111
12.2 Development Requirements . . . 112
12.3 Features . . . 113
12.3.1 System Plug-in . . . 113
12.3.2 TinyOS Environment Wrapper . . . 116
12.4 Code Analysis . . . 118
12.4.1 Scanner and Parser . . . 119
12.4.2 Extending the Parser . . . 120
12.5 Related Work . . . 121

13 Conclusion **123**

Introduction

Sensors have become an indispensable part of many industrial processes as well as our everyday lives. Whether in a power plant or a greenhouse, wherever we go we find sensors monitoring the proper operation of machinery or tracking specific environmental conditions. But also in our homes more and more sensors are employed to simplify our lives.

A limitation of today's sensing technologies is their wiring. If sensors are not directly embedded in the devices that have to respond to the sensor readings, an often cost intensive deployment of cables and wires becomes necessary. This wiring can quickly become the dominant cost factor for a deployment using multiple sensing stations. Therefore, it is often the case that a compromise between the quality of the acquired sensor data and the available financial budget has to be made.

As a practical example consider a vineyard using soil moisture sensors to optimize irrigation of the vines. As vineyards are usually situated on a slope a single sensing station does not suffice to measure soil moisture. Thus, multiple sensing locations within a possibly square kilometers large area have to be set up. Covering an entire vineyard with wires would be extremely expensive and error prone as cables may easily be damaged by agricultural machinery.

With wireless sensor network technology we can overcome both of these problems. Wireless sensor networks employ matchbox-sized micro computers offering very limited computation power and only a few kilobytes of RAM at each sensing station. These micro computers are commonly known as "sensor nodes" or "motes". The application specific sensors, in our example case soil moisture sensors, are attached to a sensor node at each sensing location. To enable data collection from the deployed sensor nodes, the devices also contain a short-ranged radio module. These radios are tuned for minimal power consumption and thus only offer a transmission range of a couple of meters to a few hundred meters under ideal conditions. In most practical deployments this results in the problem that not all sensing stations are

within direct communication range of a central base station collecting the readings. In order to circumvent this limitation, sensor networks use multi-hop communication. That is, sensing nodes closer to the base station act as a communication relay for nodes unable to reach the base station directly. Using multi-hop communication the physical dimension of a sensor network is no longer limited by the transmission range of the employed radio and also very large areas such as a vineyard can be equipped with this technology.

As its name implies, wireless sensor network technology, expects sensor nodes to be untethered. Consequently the power supply of the sensor nodes must be contained within the nodes themselves. Commonly, batteries are used as a power source. In some deployments also additional energy harvesting mechanisms such as solar panels are used to provide additional power to a rechargeable battery. Using additional energy harvesting is increasing the costs of a deployment and in general the available power a node may spend is still limited. Energy-efficiency is therefore one of the main design goals for all sensor network deployments. The communication subsystem is one of the main energy consumers of a sensor node. From an energy point of view it does not matter whether a node is receiving or sending, the power consumption is approximately the same in both cases. It is therefore vital to turn off the radio whenever possible to prolong the life-time of a sensor node. In case of multi-hop communication this raises the challenge of ensuring that both communication partners have their radios turned on at the same time to enable communication.

We have investigated the design and development of energy-efficient applications for sensor networks. We thereby worked on two orthogonal axes which are represented as two parts of this thesis. The first part is dedicated to the Dozer project. Dozer is a communication system designed for long-term data aggregation at low data rates. Applications in this domain include most environmental monitoring tasks such as our vine yard example. With Dozer, sensor nodes can operate in a multi-hop network for up to ten years on one set of non-rechargeable batteries and without any human intervention.

As wireless sensor networks are still a young field of research and so far only a small number of commercial players are working with this technology, there are also only a limited number of development tools available. Generic text editors for code development and command line based compilation tools are still the standard with which most developers in this area have to work. This lack of development support further complicates the design and implementation of new protocols and application for sensor networks. It results in a high barrier to entry for new interested developers and slows down the actual development process. The second part of this thesis therefore presents development tools we have built helping to mitigate the problems faced when building industrial strength sensor network applications.

Part I

The Dozer Project

Chapter 1

Introduction

The first part of this thesis is dedicated to the Dozer project. Dozer is a network stack optimized for ultra low-power data gathering applications in sensor networks. As the title indicates we will give a comprehensive report on this project. Thereby the in-depth analysis of the developed algorithm is in the focus. We benchmark Dozer's performance on a set of different wireless sensor network testbeds. This includes small scale tests to highlight specific details but also extensive tests on medium to large scale networks are shown. Beside this analysis we also discuss the background of the project, how it came into existence, why this class of applications is of such importance for sensor networks, and also pitfalls and lessons learned which go beyond the protocol and its implementation.

1.1 The Origin of the Dozer Project

Before we come to the actual Dozer system we take a look at how the project started and what initial goals were set. The kick-off was a discussion between Shockfish SA [51] and us, the Distributed Computing Group (DCG) at ETH Zurich. Shockfish had plans for an application using wireless sensor networks and was developing a custom hardware for this application based on their TinyNode 854 platform (also see Section 3.1). They required a new communication stack and were wondering if it was possible to develop a system fulfilling the following three requirements

- Reliable data aggregation at one or possibly several data sinks.
- Up to 10 years of network life-time on two AA batteries (limiting the possible radio duty cycle to approximately one per mill).

- Less than 2 minutes of delay before a reading reaches the data sink in a multi-hop network.

At the time when Shockfish introduced us to their application idea there was already an ongoing discussion at DCG about an algorithm combining scheduled communication and randomization. This idea seemed a promising approach for a system solving Shockfish's specific task. Hence, further meetings with Shockfish were held to define the concrete requirements. Unfortunately, due to a NDA with Shockfish we are not at liberty to name the application they had in mind. However, for this thesis the actual application is of minor importance. The nature of the logged data has no influence on the communication system as long as we know the characteristics of the network, traffic pattern, and probability distribution of the logged events.

- Each node in the network produces periodic readings which need to be forwarded to a base station.
- One or more custom sensors on each node are used to detect the change of a physical phenomenon.
- The event to detect happens comparatively seldom and non-periodic.
- Event detection is independent at each node. That is, event detection at one node has no influence on the readings of neighboring devices.
- Multi-hop communication is required.
- Communication must be reliable. Individual messages may be lost but no "burst losses" dropping several consecutive messages are allowed.
- Topology changes in the network may happen at any time, especially when the event to detect occurs.
- Battery changes are out of the question and energy harvesting is no option either.

These requirements boil down to a periodic data gathering application with comparatively low data rates but long life time demands. Delay must not be completely ignored but is not of primary concern. Thus, Shockfish's application idea falls in the category of environmental monitoring which is also one of today's main application areas for wireless sensor networks. After the initial discussion phase DCG and Shockfish started a joint venture in the form of a CTI project [6] with the goal of developing a new communication stack suited for Shockfish's application scenario. Shockfish's role was to provide the hardware and hardware support (including drivers) as well as a back-end solution. Furthermore, according to the initial plan Shockfish

was also expected to run experiments and tests on a testbed consisting of multiple TinyNodes to validate the fitness of the developed solution. In turn the actual design and implementation of the sensor network software was to be done by DCG.

1.2 Data Gathering in Wireless Sensor Networks

Observation and interpretation of natural phenomena has always been of fundamental importance to numerous research areas and industrial applications. Sensor networks represent a new tool providing the possibility to sample and gather data at scales and resolutions which were difficult to obtain before. By spreading large numbers of cheap untethered sensor nodes in an area of interest it becomes possible to monitor dense temporal and spatial data over an extended period of time. With this data the analysis of complex interactions becomes possible.

Wireless sensing devices exhibit a large variety of favorable attributes. They facilitate the deployment and are far less intrusive than tethered solutions. Furthermore, they permit temporary measurements or surveillance of secluded areas. In addition sensor networks should not need any human interaction while fulfilling their intended tasks. Due to the limited capacity of common power supplies for sensor networks, such as batteries or solar cells, energy efficiency is a fundamental requisite for prolonged network lifetime. All sensor nodes are equipped with a short-ranged radio allowing them to convey their data to an information sink for further processing. This communication subsystem is one of the primary power consumers of a sensor node. The energy wastage of the radio, even in idle listening, is three orders of magnitude higher than a node's power drain in sleep mode. As a consequence, the radio should only be turned on if a data transfer is pending. This requirement is hard to fulfill since multi-hop routing techniques must be applied to transmit data from all nodes in a possibly large area to the data sink. Energy-efficient data exchange is a nontrivial task in single-hop networks but becomes even more challenging if routing over multiple hops is required. Sensor nodes are no longer able to schedule their transmissions strictly according to their individual demands but they also have to activate their radio in order to receive and relay messages from other nodes in the network. This raises the problems of idle listening and overhearing which waste precious energy.

1.3 Design Philosophy

Dozer is designed to provide an optimal communication system for applications in the field of environmental monitoring. We focus on applications producing continuous data. Examples thereof include precision agriculture [3], glacier displacement measurements [32], natural habitat monitoring [31, 37], or microclimatic observations [56]. All of these applications generate periodic data samples at low rates resulting in light traffic load and thus low bandwidth requirements. Dozer incorporates a full communication stack including a MAC layer, topology control, and a routing protocol. We refrained from integrating existing low-power solutions for any of these subsystems since it is our strong belief that only a perfectly orchestrated network stack is able to achieve minimum power consumption and therefore maximize network lifetime. The primary goal of Dozer is to reduce idle listening and overhearing. In theory, a TDMA-based MAC protocol constructing a global schedule to determine exact send and receive times for each node would solve the problem of overhearing and idle listening. However, in a real-world setting clock drifts and frequently changing external conditions render plain TDMA costly since maintaining an accurate schedule is a complex and energy consuming task. Dozer takes these aspects into account. It builds a data gathering tree on top of the underlying network topology and provides nodes with precise wakeup schedules for all communication only relying on local synchronization. Furthermore, it addresses the problem of temporary network partition and energy efficient tree adaptation in case of local link failures.

Incorporating the functionality of link layer, network layer, and transport layer in one piece of software, Dozer violates the design concept of layering. Layering is a proven, successful approach to build a communication system. Separating different tasks in self-contained layers simplifies development and debugging and also enables replacing individual layers. Nonetheless we have decided to abandon this concept in Dozer. The reason for this decision lies in the overhead incurred by the separation of a task in multiple layers. For our specific application we had to consider the challenge that certain real-world problems could not be handled on one individual layer without the whole communication stack being aware of the changes. An example hereof is the selection of communication partners. Due to the constantly changing quality of the wireless channel all sophisticated sensor network applications have a functionality monitoring the reliability of a connection between two communication partners. As the available memory on the devices is scarce and link monitoring potentially costly in terms of energy it is usually not feasible to monitor the links to all possible neighbors of a node. We therefore face the question who should select the appropriate connections and according to which information? Traditionally, this task is solved by a "topology control

layer” placed between the network and transport layers. For a wireless sensor network the selection of the proper links is thereby influenced by multiple aspects including the reliability of a specific link, the amount of traffic to be routed over the connection, and if the connection has to be established to prevent an undesired separation of the network into multiple not mutually connected clusters. The input for the topology control component therefore stems from all layers in the communication stack and requires a lot of inter-layer communication. Here Dozer benefits from its holistically designed communication stack. For example, the link layer is aware of the existence of a topology control mechanism and thus reports connection problems directly to this component.

At this point it is important to point out that the lack of layering must not be confused with a lack of modularization. It is vital for a system to have components with individual well-defined subtasks but it is helpful if the design of the system allows components to be aware of each other. By having a component do some extra work which may not be essential for its main task it may become possible to solve challenges in other modules much more efficiently. The components are no longer strictly self-contained and may not be hierarchically placed on a stack but they still have their individual tasks. Dozer is built according to this principle as all modules are designed to optimize the performance of the entire system and not one component.

Chapter 2

Dozer System

2.1 Dozer Overview

The Dozer system is indented to meet common demands of environmental monitoring applications. It enables reliable data transfer, has self-stabilizing properties—and is thus robust against changes in the environment—, and it is optimized for long system lifetime. Network latency and flexibility towards dynamic bandwidth demands are considered to be of less importance.

In order to forward data to the base station Dozer establishes a tree structure on top of the physical network. This guarantees that information from any node is conveyed on a loop-free path to the data sink which constitutes the root of the tree. Each node fills two independent roles in tree maintenance. On the one hand, it acts as a parent for directly connected nodes one level deeper down the tree. On the other hand, it is a child of exactly one node one level higher in the tree. Data is transferred to the sink using a TDMA protocol. However, Dozer does not construct one global schedule for the whole network but splits it up at each node. Consequently, each node has two independent schedules; one in its role as a parent and one for its child role. As a parent a node decides when each of its children is allowed to upload data. Vice versa, in its role as a child it receives an update slot from its own parent. Thus, Dozer only constructs single-hop schedules and does not rely on any global synchronization. Each round of a parent's TDMA schedule is initiated by the transmission of a beacon message. Simplified, beacons are the heart beat of the Dozer system. In its child role, a node synchronizes on the received parent beacon. However, it does not adjust its internal clock but calculates the offset of its upload slot in relation to the last beacon reception time.

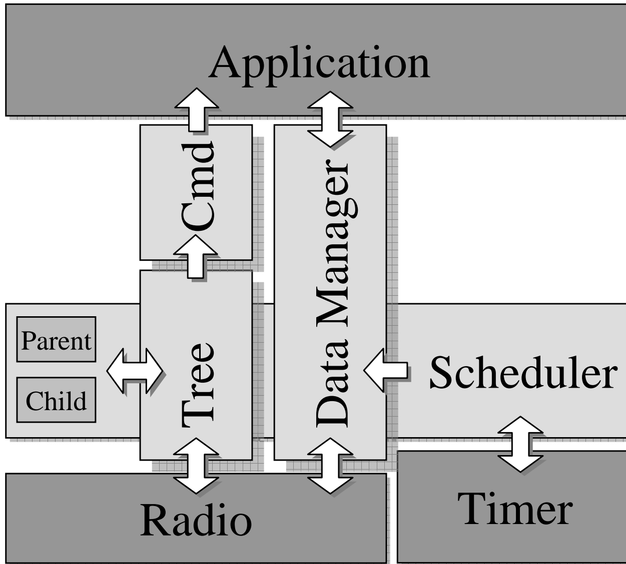

Figure 2.1: Architecture of the Dozer system represented by the light gray boxes. Arrows indicate the command flow between the different modules.

Dozer does not make use of a traditional MAC protocol. In fact, the system does not try to prevent nodes from sending at the same time; collisions are explicitly accepted (c.f. in Section 2.2.2). Using randomization Dozer ensures that two schedules drift apart quickly in case of a collision. This scheme is advantageous as a message receiver exactly knows when the corresponding sender is going to start its transmission. This greatly prolongs network lifetime as nodes are able to maximize their time in energy-efficient sleep mode. Facing collisions data transmissions in Dozer are always explicitly acknowledged.

As network conditions change over time so does the network topology. Consequently, the data gathering tree cannot be stable in the long run. To reduce increased message delay in case of link failures, each node maintains a list of additional potential parents. Choosing a candidate from this list a new connection can be established with little overhead.

2.2 Dozer Implementation

As the high-level overview in the last section has outlined Dozer handles several interwoven tasks in parallel. More precisely, the system can be subdivided into four logical components. Figure 2.1 depicts the individual components and shows how they interact with each other. In the following the function of each component is discussed in more detail.

2.2.1 Tree Maintenance

The *Tree* module coordinates a node's integration in the data gathering tree of Dozer and guarantees constant connectivity. Furthermore, in case of a network failure it sets the node in an energy efficient suspend mode until a reintegration in the tree becomes possible.

Connection Setup

It is essential for every node to be part of Dozer's data gathering tree. Nodes without connectivity are unable to provide data to the base station and are thus of no use. Upon wakeup, in the bootstrap phase, a node tries to join the tree as quickly as possible. Since it does not yet have any conception of its neighborhood, it starts listening for beacon messages of nearby nodes. Beacon messages are periodically sent by already connected nodes at the beginning of their TDMA schedule to enable the integration of disconnected nodes. After scanning for the full length of a TDMA round each received beacon message is analyzed and the corresponding node is ranked according to a rating function. The function's current implementation considers a node's distance to the sink as well as its load—the number of direct children—in this computation. Both of these values are part of the beacon message and are thus readily available. To minimize tree depth, distance has a higher weight than load in the computation. The node now tries to connect to the highest rated neighbor and the gathered information about all other overheard potential parents is stored.

The actual connection setup is initiated after the transmission of the next beacon of the selected neighbor (see Figure 2.2). After sending its beacon the potential parent stays in receive mode for a short amount of time. Within this contention window it accepts incoming connection requests. The child uses a simple random back-off mechanism to determine when to send its connection request message. This contention phase is needed since multiple nodes may want to establish a connection with this parent at the same time. On receiving a connection request message the parent assigns the new child a slot in its TDMA schedule and returns this information by means of a

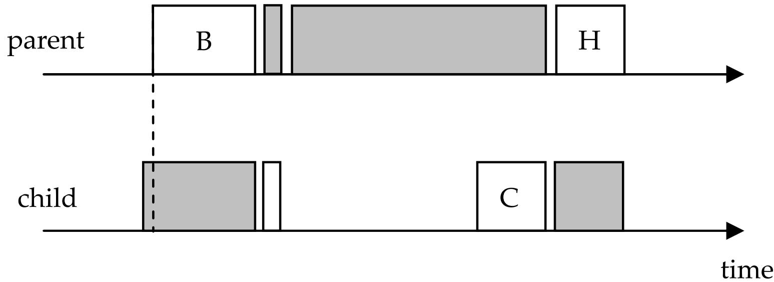

Figure 2.2: Connection setup – The parent node sends a beacon (B). Upon beacon reception the child sends a busy tone to activate the contention window. The child then transmits its connection request (C). A handshake (H) serves as an acknowledgment. Shaded areas denote the times a node is actually listening.

handshake message. Currently, a node only accepts one new child per beacon interval. This restriction serves as a simple form of load balancing. A node failing to connect to a specific neighbor may first try to join the tree at another node with similar rating before retrying the same parent.

Since listening for the whole length of the contention window after each beacon transmission is expensive, in Dozer an activation mechanism precedes the actual connection setup. As depicted in Figure 2.2 the child transmits an *activation frame* immediately after receiving the potential parent's beacon message. On the other side, the parent switches to receive mode and polls the channel right after sending its beacon. Only if the received radio signal strength (RSSI) indicates channel activity the contention phase is activated. If multiple nodes want to connect to the parent in the same round their activation frames collide. This is no problem since the parent does not try to detect a specific pattern and the sensed RSSI still clearly indicates activity on the medium.

Connection Recovery

Wireless links are fragile to changes in the environment and must be expected to break at any time. Unstable weather conditions or temporary obstacles in the area of interest can have a negative impact on the network stability. Dozer incorporates a mechanism to confront this problem.

A connection to the current parent breaks if multiple consecutive data transfers fail and the parent is declared unreachable. To replace it with little overhead, the orphaned node queries its stored list of potential parent for

a well suited substitute and tries to establish a new connection. In case of success this procedure costs a reasonably small amount of energy. However, if no replacement can be found in this list the node falls back to bootstrap mode (see Section 2.2.1) and has to conduct a costly scan in order to detect new potential parents. To guarantee the availability of reasonably up-to-date information about its stored potential parents, a node periodically listens for their beacons. This refresh is cheap since future beacon transmission times of a node can be predetermined accurately based on the point in time of its last overheard beacon. This calculation is performed by the Scheduler component described in Section 2.2.2. Additionally, to learn about the existence of new, potentially well suited parents, a random listen mechanism is applied. Unrelated to their two schedules nodes periodically overhear the channel for beacons of yet unknown nodes. To keep the incurred overhead low, these scans must only be executed infrequently.

Suspend Mode

If a node is not connected to a parent and also cannot hear any beacons—even when listening for a full beacon interval—it assumes the network to be down or out of reach. Constant channel surveillance in this situation would result in high power consumption and a node's lifetime would decrease to a couple of days. To circumvent such energy wastage Dozer features a special suspend mode. Along the line of low power listening [42] the node periodically samples the channel for activity and remains in sleep mode for the remainder of the time. However, unlike low power listening this mode does not ensure reception of all messages. Energy efficiency and quick reactivation in case of channel usage can be balanced depending on the demands of the application running on top of Dozer. Frequent channel polling results in higher power drain but also more rapid connection establishment on network availability. On the other hand, longer intervals between scans lead to improved energy efficiency but possibly delayed reintegration of suspended nodes.

2.2.2 Scheduler

The energy efficiency of the Dozer system mostly stems from the *Scheduler* module. By providing the Tree Maintenance and Data Manager modules with precise timings it enables efficient radio usage.

Communication between a parent and its children is coordinated by a TDMA protocol. That is, all transmissions happen at exactly predetermined moments in time. For the exchange of a message neither sender nor receiver have to spend energy beyond what is required to transmit or receive the actual data. In particular nodes do not have to waste energy on overhearing

the channel for pending transmissions. However, a global TDMA scheme is expensive since it demands the existence of a network-wide time synchronization mechanism. To circumvent this burden Dozer only aligns one hop neighbors in the data gathering tree. As all nodes are simultaneously parent and child they all have to maintain two schedules; one provided by their parent and one self-determined as a reference for their children. In this setting it is complex to synchronize the internal clocks of a parent and its children. Only by means of global time synchronization it would be possible for each node to service both schedules with only one clock.

While in theory wakeup times can be calculated perfectly at both parent and children, clock drift has to be considered in real-world applications. The current generation of sensor nodes is usually equipped with an electronic oscillator exhibiting a skew of 50 parts per million (ppm) at room temperature. Thermal differences between sender and receiver lead to significant, additional skew. In Dozer, the receiver of a transmission is responsible for clock drift compensation and worst-case guard times are used to guarantee a prior wake up of the receiver before the sender starts its transmission.

The self-determined TDMA schedule of a node, in the following also denoted as parent schedule, is of fixed length and divided into equal time slots. Upon connection of a new child the Tree Maintenance module requests a free slot from the Scheduler. This slot is henceforth marked as occupied and reserved for the new child. The assignment outlasts the end of the schedule and is only released if the corresponding child disconnects. That is, each child owns the same time slot in every iteration of the schedule. As a consequence, the total number of slots of the TDMA schedule defines the maximum number of children a node is able to manage.

After connection establishment between parent and child, the personal slot number of the child in its parent's schedule is known at both nodes. They can thus compute the start of this slot relative to the beginning of the schedule. For each slot of its schedule the parent checks if it is occupied and listens for incoming data if necessary. Analogously, at the child the Scheduler triggers the Data Manager component at the start of its upload slot to permit a timely transfer of potentially queued data messages.

As mentioned above the protocol does not provide any direct clock synchronization. Instead, at the outset of a new round of the schedule the Tree Maintenance module is triggered to send a beacon message. This beacon is received by all children and timestamped according to their local clocks. Since both parent and children share the knowledge about the time of the beacon transmission this moment in time serves as an anchor point for implicit clock synchronization. No adjustments of system clocks are required but only this timestamp needs to be stored for further timing calculations. For the remainder of this round of the TDMA schedule all events are computed in

relation to this timestamp. The transmission time of the next beacon is also determined according to this value. As a positive side effect, clock drift accumulation over multiple rounds of the schedule is prevented. Furthermore, the complexity of handling both independent schedules diminishes since only two values used as offsets for the internal clock need to be stored.

Without a global schedule, collisions between the transmissions of neighboring nodes that are not part of the same schedule can no longer be excluded. Other systems facing the same problem (e.g. [19]) apply secondary MAC protocols such as CSMA/CA to resolve it. However, since bandwidth demands in the considered scenario are low, collisions happen infrequently. Dozer thus refrains from handling them actively. In the long run the costs for retransmissions are cheaper than the costs that would arise to prevent them. But regarding collisions there exists an additional problem which needs to be tackled. Collisions may indicate the undesired alignment of two independent schedules. If this is the case, without intervention, collisions would recur in subsequent rounds of the schedule. To counter this threat, Dozer extends the length of a TDMA round by a randomly chosen time span—also referred to as jitter. The parent draws a new random number for each round of the schedule which is then added to the round's length. There is a linear relation between the maximum transmission time per slot and a reasonable upper bound for this random offset. Dozer uses a bound of seven times the time needed to flush the local message buffer (c.f. in Section 2.2.3). With this value, in case of a collision between two unsynchronized transmissions, the chance for a second consecutive collision is less than 50% in expectation. For any realistic scenario this implies that a maximum jitter of less than one second suffices.

This random prolongation of the TDMA rounds introduces the challenge of how to predict the exact time of the next parent beacon. Thus, the seed value of the random number generator used for calculating the next random offset is included in every beacon. With this value each child is able to execute the same computation as the parent and to predict when the next beacon message is due. At the parent, the current random number is used as seed value to generate the next random number. Consequently, even if a child misses one or more consecutive beacon messages of its parent it is still able to determine the next beacon arrival time: Based on the information of the last beacon it has received, it recursively draws random numbers until it has compensated for the number of missed beacons.

2.2.3 Data Administration

At the end of the day, Dozer's main task is to transport sensor readings from all nodes to the data sink. While a node's data upload times are strictly

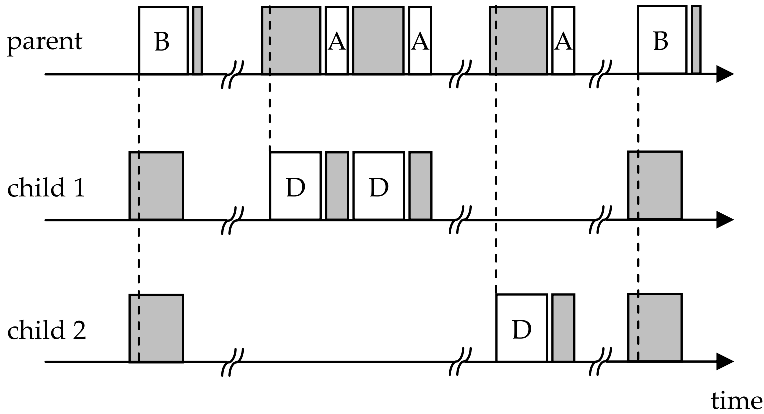

Figure 2.3: Message reception of a parent with two children. Upload slots are determined by parent beacon (B). All data messages (D) are explicitly acknowledged (A).

defined by the Scheduler module, data injection by the application is always possible. Hence, the *Data Manager* module features a message queue buffering injected data pending for transmission. Since data upload to the parent and data reception from the children is unsynchronized, incoming messages from the children are also appended to this queue.

As soon as the Scheduler module signals the beginning of the parent upload slot the Data Manager tries to transmit all queued messages. Each message is explicitly acknowledged and only removed from the queue of the sender if the receiver confirms its correct reception (see Figure 2.3). With the acknowledgment the parent not only takes over responsibility for the packet but also notifies the child about how many additional messages it is willing to accept. Consequently, at most one unnecessary message transmission is possible if the parent is unable or unwilling to handle more messages. The link acknowledgments guarantee that no messages are lost on their path towards the sink despite possible collisions on the wireless links. If a message transfer fails to be acknowledged the child immediately stops its data upload for this round of the schedule since a temporal interruption on the medium may be encountered. In case of consecutive transmission failures over multiple upload slots the Data Manager instructs the Tree Maintenance module to switch to another parent. Due to the limited amount of memory available on current sensor node platforms the queue size is limited. Different buffering strategies

may be employed depending on the application requirements. Dozer's default strategy only allows buffering of one data message from each distinct node; if more than one message from the same origin meet on a node the newer one is buffered and forwarded whereas the older one is discarded.

2.2.4 Command Management

While data flow in Dozer is strictly unidirectional towards the sink it is often desirable to be able to send information to one or several nodes in the network. Dozer establishes a lightweight back channel by making use of the beacon messages. Commands injected at the data sink are piggy-backed on its next beacon message. Every node receiving a beacon containing a command temporarily stores the command and includes it in its own next beacon. By repeating this procedure at each level of the tree the command is disseminated through the whole network. Besides addressing a command to all nodes in the network the injection of commands for individual nodes is also supported. Nodes not directly addressed by a command still relay it to enable propagation to nodes deeper down the tree.

Upon reception of a beacon message from the parent the Tree Management component hands the command to the *Command Manager* module for further processing. The module checks if this node belongs to the set of intended recipients of the command. If this is the case the command is dispatched to the application running on top of the Dozer system. Thus, applications are able to define their own custom commands and corresponding command handlers.

Chapter 3

Evaluation and Testing

In this section we evaluate Dozer's performance under different conditions in real-world testbeds. First, a set of preliminary measurements on a small-scale network are conducted to estimate the scalability of the system. In a second step we present results of a deployed indoor network consisting of 40 sensor nodes.

3.1 Hardware and Operation System

For all experiments we used the TinyNode 584 sensor platform [17] produced by Shockfish SA. It features a MSP430 mirocontroller with 10 kB of RAM and 48 kB of program memory. Furthermore 512 kB of external flash memory are available. However, due to the high energy costs for flash access Dozer does not make use of it. The platform includes a Semtech XE1205 radio transceiver. This radio is known for its good transmission ranges and high

	Current Draw	Power Consum.
uC sleep , radio off	6.0 uA	0.015 mW
uC active, radio idle listening	12.17 mA	30.43 mW
uC active, radio RX	12.63 mA	31.58 mW
uC active, radio TX	16.10 mA	40.25 mW

Table 3.1: Measured current consumptions of the TinyNode platform in different states at 2.5 volt.

Figure 3.1: TinyNode 584 with extension board

data rates of up to 153 kbit/s. For our measurements the nodes were operated at 868 MHz using 0 dBm transmission power and a bandwidth of 75 kbit/s[1]. As a power source two customary 1.2 volt rechargeable batteries were installed with a capacity of 1900 mAh each. The measured current draws for sleep mode, idle listening, receiving, and sending under these conditions are shown in Table 3.1. As can be extracted from the table, on the TinyNode platform idle listening is nearly as expensive as the actual reception of a message. Thus it benefits greatly from Dozer's scarce use of unscheduled random channel overhearing. Furthermore, the cost for transmission and reception of a message are in the same order of magnitude.

Dozer is implemented on top of the TinyOS-1.x operating system. No changes were made to the operating system excepts the replacement of a timer module whose genuine version contained a bug. Under certain conditions timer events are triggered too late. In normal operation common TinyOS-applications do not encounter this undesired behavior frequently. However, due to Dozer's intense use of the timer module, this malfunction regularly occurs in our system with disastrous consequences. A once deferred schedule becomes useless since all relative timings are out of sync. Consequently, a node affected by this problem inevitably looses connectivity and

[1] As described in [17], at the same transmission power, the XE1205 radio attains higher communication ranges than other state-of-the-art platforms. Hence, we are able to transmit at lower power while still achieving good ranges.

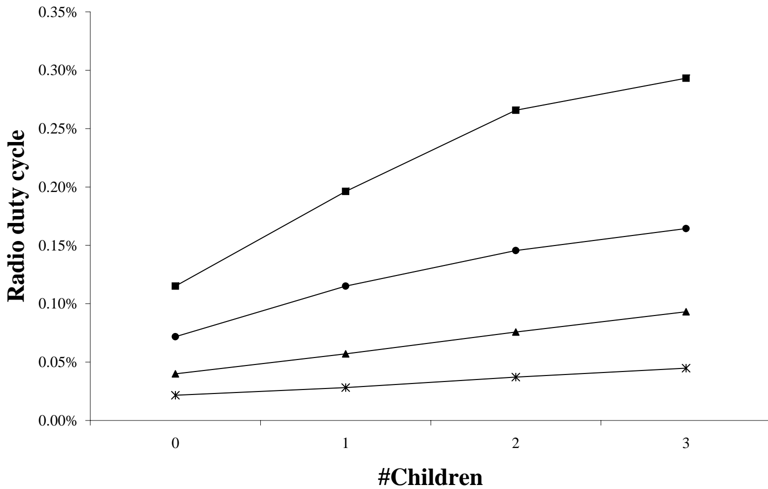

Figure 3.2: Radio duty cycle of a node depending on its number of children. Measurements were performed with beacon intervals of 15 s (square), 30 s (circle), 1 min (triangle), and 2 min (star), respectively.

falls back to bootstrap mode. Thus the replacement of the timer module was mission critical.

The memory footprint of Dozer is 20 kB in program memory and 1.7 kB RAM. The message queue of size 20 in the Data Manager module used as a temporary buffer for messages which need to be relayed thereby contributes 39% of the RAM usage.

3.2 Small Scale Experiments

Measuring the energy drain of a node is a non-trivial task. On the one hand, the measuring interval is too long for high-resolution measurements with an oscilloscope. On the other hand, a voltmeter is too inaccurate to capture short changes in current draw. Hence, we decided to measure energy consumption indirectly. For this purpose, all nodes log their radio duty cycles. This is achieved by summing up the differences between radio startup and shutdown times. Since spotting the exact switching times from send to receive mode and vice versa is difficult, only the total uptime is recorded ignoring the specific state of the radio. This information is propagated to the base station using Dozer's own data gathering mechanism. The collected

information can be converted to power consumption values using Table 3.1. As nodes only provide the overall radio uptime, a worst-case approximation is made. That is, it is assumed that they are always in transmit mode if their radio is active. As a consequence, all further results related to energy consumption can be considered as an upper bound for the actual power consumption.

To investigate the relation between a node's power drain, its number of children, and the beacon interval time we conducted a series of experiments on a small network with predefined topology. In each run the node of interest was directly connected to the sink. Over time, up to three children were included in the network and forced to connect to the monitored node. This sequence was repeated with beacon intervals in the range of 15 seconds to two minutes. The data sample interval was set to four times the length of a beacon interval. The results of these experiments are depicted in Figure 3.2.

Originally, the goal of this experiment was to come up with lower bounds for the achievable duty cycles at different positions in the data gathering tree. However, test results exhibited unexpected fluctuations when run with different sensor nodes. After closer examination, it became clear that the inherent clock drift within a single beacon interval is a significant factor influencing the total duty cycle. Thus the following results do not represent precise lower bounds. Nevertheless, they provide an accurate approximation of the radio uptimes in real networks.

Figure 3.2 shows that the duty cycle decreases as the beacon interval grows larger. This elementary observation is based on the fact that the number of messages to be transmitted within one beacon interval is constant independent of its length. Hence, longer intervals lead to prolonged sleeping periods without significantly increasing the radio uptime. Using a similar line of argument, the variable additional costs for a newly connected child at different beacon intervals can be understood. The reduction of the incurred overhead for the fourth child in the 15 second beacon interval experiment

Beacon interval	30 s
Max. jitter	650 ms
Data sampling interval	120 s
Potential parents update interval	15 min
Overhearing	1 s/4 h
Compensated clock drift	100 ppm
Max. stored potential parents	5
Message queue size	20

Table 3.2: Configuration of the Dozer system for the office floor testbed.

illustrates another phenomenon worth mentioning. Costs for additional children do not necessarily have to grow linearly. Simplified, a parent's costs for a child are twofold. On the one hand, it has to receive the child's data messages. These costs cannot be prevented. On the other hand, the parent has to forward the received messages. Thus, in its next upload slot it has to power up the radio and send the pending messages. Since the radio start-up and shutdown consumes a similar amount of time ($\sim$ 2 ms), and thus energy, as the transmission of an actual data message ($\sim$ 5 ms) its overhead is not negligible. Consequently, if the parent is able to upload data from two or more children in one upload slot it saves the additional overhead of turning on the radio for each of these children individually—that is, costs per child decrease.

3.3 Office Floor Experiment

To put Dozer's fitness for real-world deployments to the test, a generic indoor network was run for several weeks. The topology was no longer predefined for this setting but automatically constructed by the Dozer system.

3.3.1 Setting and Protocol Parameters

The considered testbed consisted of 40 TinyNode sensor nodes which were deployed on one floor of our office building (see Figures 3.4 and 3.3). The dimensions of the building are approximately 70 x 37 meters resulting in an testbed area of more than 2500 square meters. During the whole operation of the network, the floor was populated by more than 80 people during office hours. Thus the nodes where exposed to frequently changing environmental conditions. Furthermore, during the deployment phase special attention was paid to construct a network with heterogeneous density. While nodes were concentrated in the upper-right part of the building to achieve a dense region, the southern part was only sparsely populated. This allowed the performance evaluation of Dozer in networks featuring different characteristics. In addition to 38 sensing nodes, a base station (Node 0) was placed in the upper-right corner of the map. This location was chosen to get a deep data gathering tree and to enforce multi-hop communication. One further extra node was positioned in the vicinity of the sink for debugging purposes. This node acted as a network sniffer which overheard and logged all network traffic at the base station.

In total Dozer was tested for more than one month on this network. Detailed logging information forming the basis of the evaluation in this section were gathered during one week of operation. Each node thereby sent ap-

Figure 3.3: Left: During the benchmarking of the Dozer system our office building was under heavy construction work leading to additional interference. Right: To increase the communication range of the base station it was mounted on a cable rack about 2 meters above the floor.

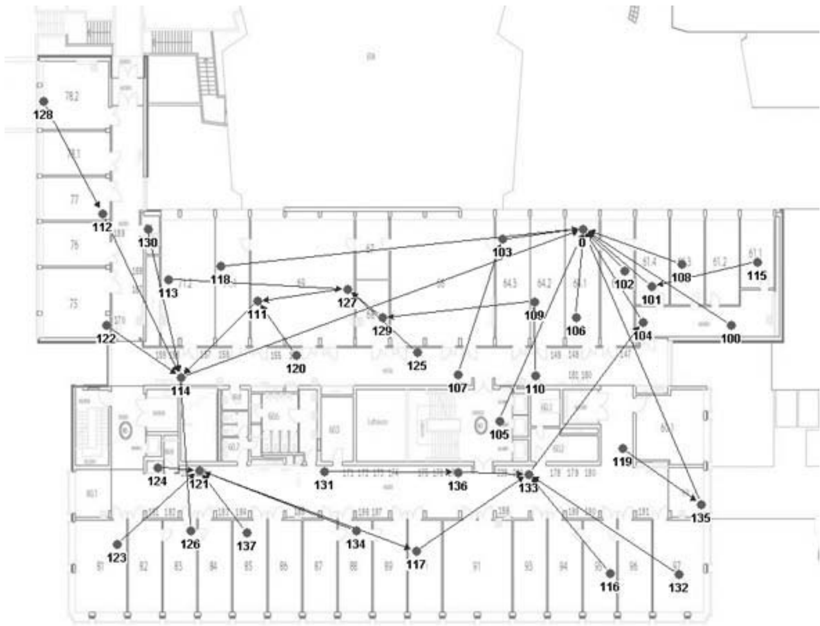

Figure 3.4: Indoor deployment of 40 sensor nodes including a snapshot of Dozer's data gathering tree. Node 0 (upper-right corner) acts as data sink.

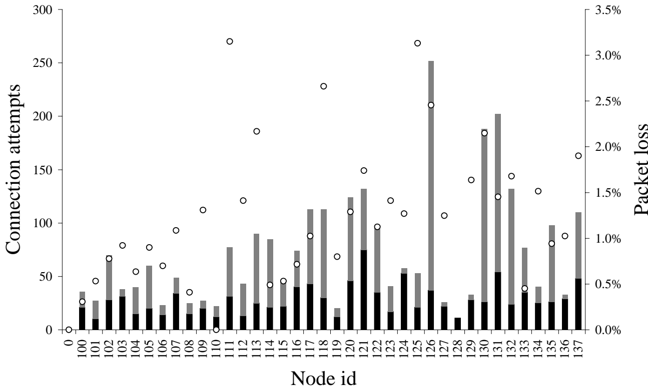

Figure 3.5: Number of successful (black) and failed (grey) connection attempts per node. Per node packet loss on the second y-axis.

proximately 5000 data messages to the sink. As described in Section 2.2 the Dozer system can be tweaked to suite the requirements of a specific application. Table 3.2 shows the important parameters and their assigned values for the office floor testbed. Although the anticipated clock drift in our scenario is less than 50 ppm, Dozer was configured to allow for 100 ppm. Consequently, more energy than strictly necessary was consumed. In return, with these settings, the system is also expected to operate properly in outdoor environments facing moderate temperature changes. All other values were chosen to represent a possible demand of a real application in the domain of environmental monitoring.

3.3.2 Tree Topology

Figure 3.4 shows a snapshot of the data gathering tree as it was witnessed during the experiment. Each node features one outgoing arrow pointing to its parent. It can be seen that the base station (Node 0) has numerous children. This has two different reasons. On the one hand, the parent rating function described in Section 2.2.1 promotes connections to the sink since the latter has zero tree depth. As a consequence, each node receiving a sink beacon first tries to connect to the base station before inquiring any other nodes. On the other hand, the base station was flashed with a slightly modified version of Dozer. Since the sink usually runs on external power—as it is also the case in our setting—it is less compelled to economize on its energy resources. Thus, the contention window was extended and the sink was configured to

accept more than one child per connection phase.

Another observed phenomenon is the fact that hardly any connections passed the central core of the building. We assume that multiple sources of interference led to this barrier. For one, the corridors are lined with solid metal lockers perturbing most radio communication. On the other hand, this zone also comprises the ventilation system, sanitary facilities, and multiple elevators producing additional interference.

We examine the stability of the data gathering tree by investigating topology changes and message loss. Topology changes are indicated by a node changing its parent. Both of these values are depicted in Figure 3.5. As hoped for, message loss was low, on average 1.2% and at maximum 3.15%. However, Node 128 is excluded from this analysis. Due to its peripheral position in the network it was only able to connect to one single other node (Node 112). In case of a temporary interruption in the connection to its parent the node went to suspend mode. In addition, the low network density in its vicinity resulted in a low probability for a quick recovery. Thus, the node suffered from message loss of approximately 30%.

The measured high message yield at the base station is evidence of the correct operation of Dozer's Tree Maintenance module. As emerges from Figure 3.5, a significant number of topology changes were necessary to cope with momentary, local channel irregularities.

3.3.3 Energy Consumption

As in the small testbed described in Section 3.2, energy consumption of the deployed nodes were measured indirectly via their duty cycles. Figure 3.6 depicts the average radio activity of each node in the network. The upward error bar shows the root mean square (RMS) error of all measurements exceeding the average duty cycle; the downward error bar is defined accordingly. The overall average duty cycle of all sensing nodes is 0.167% with a standard deviation of 0.0004. Applying the values from Table 3.1, results in a mean energy consumption of 0.082 mW.

Looking at individual nodes, the sink had by far the highest radio uptime of almost 1%. This is not surprising since it had to process the data of the whole network. Additionally, the extended contention window directly affects its duty cycle and explains the considerable difference in comparison to the sensing nodes.

Node 124 exhibits a radio uptime of 0.28%. Figure 3.7 shows a snapshot of 2000 consecutive data messages of this node. As can be seen for most of the time the node ran at a duty cycle of 0.07%. Comparing this value to the results from Section 3.2 leads to the conclusion that the node is a leaf in the data gathering tree. However, three different energy intensive

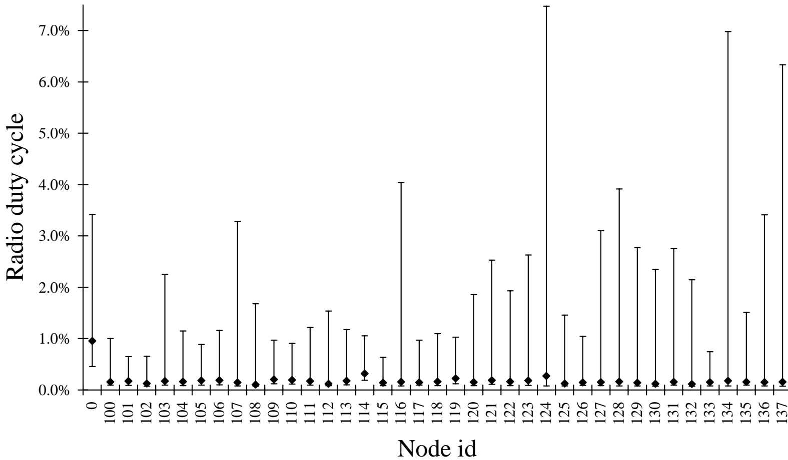

Figure 3.6: Average radio duty cycle of each node including RMS errors.

effects can be observed. First, the most dominant peaks exceeding 20% are scans for a full beacon interval. This means that the node was forced to establish a new connection but did not find an appropriate potential parent in its cache. Second, the overhearing phase once every four hours results in a temporary duty cycle of around 1%. Finally, the potential parents updates lead to the fringes of up to 0.1%. These insights and the fact that Node 124 was located in a small storage room allows the conclusion that it only had a small neighborhood. Consequently, in times of normal operation it was able to run at nearly optimal duty cycle. However, in case of connection interruptions the interference affected all its possible connections resulting in a fallback to bootstrap mode. Unlike Node 128, it only suffered from brief network disconnections. Thus, it quickly managed to reintegrate in the data gathering tree.

Node 114 features a similar average duty cycle as Node 124, namely 0.32%. But its power consumption is caused by other reasons as the different RMS error values indicate. Figure 3.8 depicts the radio duty cycle of Node 114 over a period of 2000 successive data messages. Parents updates and overhearing which are always part of a node's normal operation can also be spotted in this chart. However, there is no evidence for a bootstrap phase. In fact, Node 114 acts as a relay for several children and thus cannot reach minimal duty cycles as low as Node 124.

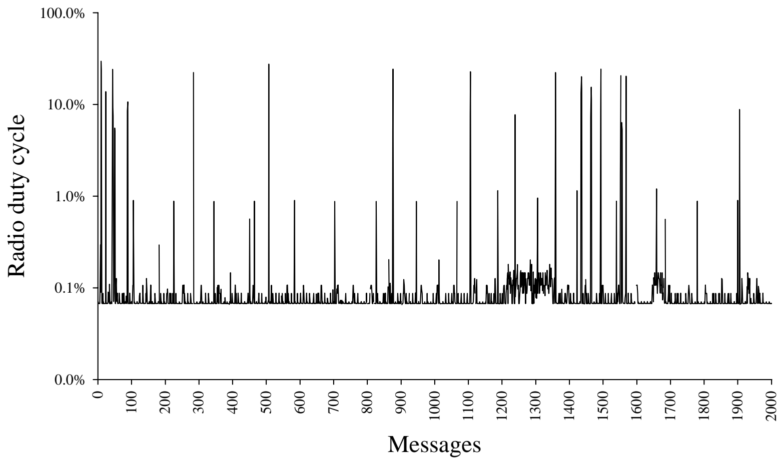

Figure 3.7: Radio duty cycle of Node 124 over a period of three days.

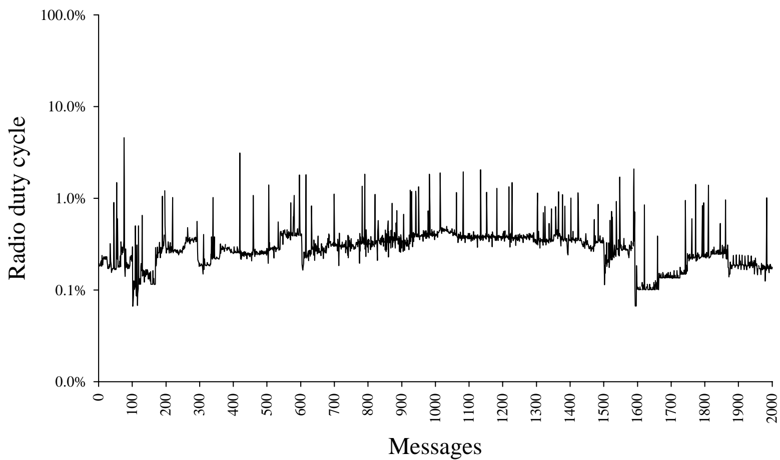

Figure 3.8: Radio duty cycle of Node 114 over a period of three days.

Chapter 4

Lessons Learned

4.1 Testing Cannot be Outsourced and Requires Adequate Testbeds

Initially it was planned to share the development process of Dozer between DCG and Shockfish. The idea was to have DCG design and implement the necessary protocols. In turn Shockfish agreed to run tests on a realistic testbed deployment. We soon learned that this plan was unfeasible. The problem turned out to be a very slow debugging cycle resulting from the outsourced testing.

We developed the code and tested it on small, mostly tabletop setups consisting of a couple of nodes. Once the system seemed OK we sent the code to Shockfish who flashed their testbed with the new release. As expected bugs lead to unforeseen behavior of the system which Shockfish then reported back to us. And here the problem of a remote testing site 200 kilometers apart from the office of the developers hit us. While both Shockfish and DCG tried to cooperate as efficiently as possible it quickly became clear that we as the software developers had to exactly specify what and also how experiments had to be monitored in order to get all necessary information. Unfortunately, this only helped partially as without detailed knowledge of the testing site it was very difficult to make assumptions on the reasons for certain behaviors of the system.

In order to speed-up development, Shockfish and DCG agreed to migrate more of the testing to our offices in Zurich. For this we set up multiple testbeds ranging from 10 to 40 sensor nodes spread over multiple rooms, or the entire floor of our office building in case of the 40 node setup. Having a

proper testing site tremendously helped in the development process as it now became possible to react much faster to misbehaviors of the system. It also made it easier to run tests under specific environmental conditions including, for example, different room temperatures or controlled external interference.

4.2 Meaningful Tests Take Time

The development time of a first "working" release of Dozer was only a few weeks. Compared to applications written for personal computers, programs for sensor networks are compact and even complex systems such as Dozer seldomly exceed 5000 lines of code. The first rounds of code debugging were therefore fast, as the obvious errors were easily found and even logic bugs in the protocol were quickly discovered. However, the way from the first release free of obvious bugs to a stable, optimized release was long and tedious. Many problems occurred sporadically and were often hard to reproduce. Such problems could not be detected by short experiments but required a testbed to run for weeks. During this time of operation any strange behavior of the system had to be tracked and inspected. It is a natural tendency to ignore minor problems, especially if the system manages to handle them and automatically reverts to normal operation. Nonetheless it is essential to follow all traces of misbehavior in the system or there is a significant risk that the cause of this minor problem may also lead to major problems in another execution. An example of such a problem we faced included nodes stopping to respond without a visible cause. The first couple of times when we saw this problem, the misbehaving node re-joined the network after a couple of minutes and resumed normal operation. The node did not reboot and did not detect that it was offline for quite a while. We first expected external interference to lead to this problem and decided to ignore it. In a longer execution of several days we also saw this problem but this time nodes did not report back for up to a day. Once again there were no anomalies before the nodes died. Everything looked perfect and then stopped working. In the end we discovered the problem to originate in a faulty timer library which returned bogus values under some very unlikely conditions. The probability to find such problems in short term tests is low and an appropriate execution time for testing is therefore vital.

4.3 Know Your OS and Hardware

This experience taught us an important lesson: When programming sensor networks one is very close to the hardware. Although the operating system tries to provide a minimal abstraction layer an application running on a

sensor node is much more vulnerable to hardware events than a PC program. Our timer issue for example was only triggered if a specific hardware interrupt occurred during the execution of another interrupt handler. To know what is going on in the operating system is the only way to discover such problems. There are attempts at relieving application programmers from this burden, for example by running a simple virtual machine on the node. However, the problem will always remain to a certain extent as the resource constraints of sensor network hardware do not allow to build arbitrary complex operating systems, protecting applications from low-level hardware events.

Another important aspect of debugging sensor network applications is the possibility to physically measure events directly on the sensor node hardware. For example, by means of an oscilloscope it is possible to detect the execution time of specific operations on the node such as turning on the radio and sending a message. Such values have to be known in order to optimize the operation of the program. If the timings are too loose a lot of energy is wasted on keeping the node in high-power states although there is nothing to do. If the timing is too aggressive the program will not work properly as the hardware may be turned off before the completion of the current task. As a consequence, if an application has to be optimized it is unavoidable to also do some measurements directly on the hardware. Besides the very high precision of this method of monitoring a node it also has the advantage that it is much less intrusive towards hardware timings than a software monitoring tool as it does not interfere with CPU usage or communication on the bus.

4.4 You do Not Get the Debug Information You Want

There is an array of options how to debug an application on one sensor node. Using the JTAG interface the exact execution of the application on the CPU can be monitored, by means of an oscilloscope the timing of many hardware components can be observed, and using the serial port even "printscreen" debugging is possible to a certain extent. Unfortunately, only the most trivial bugs can be found on one single node. In general, the complexity of a sensor network application stems from its distributed nature. Thus, the trickiest situations are very hard to reconstruct as they do not happen on one node but are a result of a sequence of events occurring an multiple nodes.

It is not possible to have a deep introspection in the state of all nodes in the network and thus only external information such as communication logs can be used to reconstruct the conditions under which a problem occurred. Nodes in the network may be executing different parts of the application code and thus be in any arbitrary state. This makes debugging more demanding as

the entire application has to be cross-checked for potential interference problems. Despite the limited size of the applications this leaves a large number of possible combinations which have to be checked. Furthermore, it is often the case that nodes do not nicely start to generate incorrect behaviors which can be logged but suddenly stop operating without any previous warning. In this case debugging happens mostly blindly as the only available information is the fact there is something wrong.

So far we have not come up with an idea how to alleviate this problem. Having multiple tests logging the same problem may allow detecting a pattern in the communication or execution of the application preceding the problem but this is not necessarily the case. For this reason it is also extremely hard to predict the time necessary to test a sensor network application before release. A problem which occurs very rarely but leads to crashing nodes can easily stall development for months.

4.5 Consider the Deployment and Offline Phases

A couple of months in the project the question of the deployment phase came up. Unlike the testbeds we worked with at that time, the final deployment site required some construction work. Therefore, it was not clear how long the physical deployment would take and in what order nodes would be installed. We had to expect some nodes to be installed several days before the rest of the network came online. This turned out to be a problem for our system. It was designed to optimize schedules under the condition that there is a network. However, in case of no connectivity nodes would continuously scan the wireless channel for activity and try to find a network to join. This was a desirable behavior for testbeds as it allowed newly updated nodes to quickly find the other nodes of the testbed. For the real-world deployment this feature turned into a serious threat. If a node was deployed a week in advance of the rest of the network its batteries would be mostly depleted before the actual operation of the network would start. So we had to come up with a solution for the deployment phase and chose a comparatively simple randomize system as described in section 2.2.1. This simple startup protocol is not optimal but allows reducing the amount of energy spent during disconnection to a level which is comparable to normal operation. It also has to be kept in mind that a node may not only be cut off from the network during the deployment phase but also during normal operations. For example a defect at the base station or essential relay nodes may lead to a disconnection of the network and also during this time, nodes have to save as much energy as possible.

Summarized, the incorporation of a protocol managing offline phases is

a key requirement for any energy aware communication system. Without such a component any unforeseen interruptions of connectivity may lead to a drastic reduction of network life time.

4.6 Decentralized Self-Organization and Self-Repair are Vital

In all of our experiments we have witnessed network instabilities. Links came and went all the time. While on average over the whole network a link might remain stable for several hours the deviation of link lifetimes was high. Especially nodes in areas with a lot of external interference (e.g. close to an elevator) or at the border of the network often had difficulties to stay connected. As a result these nodes changed their position in the network every few minutes before stabilizing for some time.

This observation was a confirmation of our expectations we had from the start. We had already predicted this behavior during the design phase of the protocols and therefore included self-organization capabilities in the network stack. Nonetheless, it was impressive to see how the network could fundamentally change its structure within a few minutes to bypass a local zone of interference. Only a decentralized algorithm can achieve such an optimization at very low power consumption. A centralized solution would first have to collect information about the current interference situation and possible connectivity of the whole network. In a next step it would then centrally compute an optimal topology. This task is very difficult to achieve in a low-power communication system as nodes close to sources of interference may not know how to route their information towards a central authority if their old communication neighbors are no longer reachable. Furthermore, a centralized system also faces the problem that by the time the optimal solution is ready to be broadcasted to all nodes, the network may have changed again and the new topology does not allow communication along the planned links.

In contrast, a decentralized solution will often not construct the optimal network structure as many decisions are taken with only limited, local information. However, it can react faster to interruptions as nodes suffering from interference problems immediately check for alternative routing options. Consequently, a decentralized system is much more robust and more apt to cope with hostile environments than a centralized solution. Message routes may become longer than on an optimal topology but communication in a "working" network is much cheaper than if bootstrap protocols have to be triggered to reconstruct a new topology. Therefore, the price for additional message relaying is more than compensated by avoiding network breakdowns.

Based on these observations we come to the conclusion that centralized

communication systems may perform very well under controlled conditions and on testbeds. However, it is questionable how they behave in real-world scenarios where interference is the rule and not the exception.

4.7 Include a Software Update Option

A large number of iterations of program code are necessary before it reaches a stable state. Approaches such as extreme programming and code audition may help reduce the number of necessary cycles but in the end testing on deployments is unavoidable.

As network sizes exceed desktop-size a pragmatic challenge arises: How to reprogram the nodes with a new software release. Traditionally, a node is connected to a personal computer using serial, USB, or similar connections and flashed with a new firmware. This is no big deal for a small number of nodes but already a setup of 20 devices easily takes 30 minutes to update. If the devices are spread over a larger area such as an office floor, time goes up significantly as the devices have to be found, collected, flashed, and redeployed. Even minute errors in the code like an incorrectly set parameter or constant can thus lead to several hours of work. If the deployment is on a remote site, the necessary effort for a software update quickly turns into days.

Academic testbeds such as [60] can feature tethered connections to all nodes or use a combination of wired and wall-powered, wireless technologies [16] to provide direct access to all deployed nodes. Usually the monitored sensor nodes are thereby attached to the testbed infrastructure using USB or Ethernet connections. If the employed sensor node does not provide a corresponding interface, additional hardware extension boards have to be used. Such a setup is cost and maintenance intensive and thus usually limited to indoor deployments of a couple of dozen nodes. Furthermore, one has to keep in mind that hardware extensions or code on the node used for monitoring purposes may change the behavior of the monitored node.

One of the most ambitious deployment support tools is JAWS [8]. Instead of a LAN connection a second sensor node or support node is attached to the primary node running the newly developed application. The support nodes form a secondary, independent wireless network over which new versions of the software can be distributed and flashed to the primary nodes. This approach removes the requirement of wiring all network nodes and is thus very flexible to set up. A remaining drawback of the JAWS system is the possible interference created by the wireless technology used on the support nodes, which may have an impact on the execution of the new application. Furthermore, the system is only intended for relatively short-term development

purposes and cannot be used for deployments running over a long time period as it is not optimized for low power consumption.

Also for a stable deployment an update option should always be considered. It may be that the requirements posed to the network change over time (e.g. data samples at a higher rate may be required) or a hidden bug is only found after several months of live operation. If collecting the nodes is no option (e.g. due to an inaccessible deployment site) and support installations such as JAWS are not possible, the only remaining option to allow software updates is over-the-air reprogramming of the nodes. In TinyOS, Deluge is a library offering this service. Unfortunately, Deluge's mode of operation does not work well with the tightly synchronized communication model in Dozer. Deluge uses a randomized announcement system in which nodes periodically broadcast information about their currently running application and other stored firmware images. If a node overhears such a message and detects that its own data is older than the announced image it will request the newer data from the announcing node and update its cache.

Since in Dozer nodes spend virtually no time in overhearing mode the random announcement mechanism of Deluge fails. We have implemented two options how to integrate Deluge in Dozer. The first approach is to store a second software image on the node in which it participates in the Deluge protocol but not in any other activities. A reboot command is flooded through the network using the command field in the beacon message to tell the nodes to boot into this update image. A time-delayed execution of the reboot reduces the risk that a node misses the reboot since the command is flooded multiple times through the network. Once all nodes have booted into the update firmware, the new program is injected in the network and spread by Deluge. A final reboot command returns the nodes to normal operation, executing the updated firmware.

The second approach involves overriding the radio control component in Dozer. Instead of returning to sleep mode directly after each transmission the radio stays powered and can thus overhear the wireless channel for Deluge messages. Once again the command field integrated in the beacon messages is used to activate this mode if an update is to be injected. The advantage of this solution is that normal operation of the network can be maintained for the whole time of the software distribution up to the point of the reboot into the new release. The disadvantage is a more complex management of the radio component and thus an increased risk that the update mechanism may be prevented from proper execution due to a bug in the application running on the node.

In both cases a software update comes with a certain risk as Deluge lacks some fundamental functions. For example, it is not possible to collect information about the update progress of all nodes. It is therefore at

the developer's discretion to decide when to broadcast the finalizing reboot command in order to make the nodes load the new firmware. If this reboot command comes too late nodes spend too much energy on idle listening, as in Deluge mode they cannot go to sleep. If the command comes too early and not all nodes have updated their stored images these nodes are unable to load the new firmware and, if a change to the communication protocols was made, are no longer reachable. Deluge is not ideal but the best option TinyOS programmers currently have to update their software over-the-air. A number of recent work [59, 52, 40] has proposed improvements to Deluge and it only seems a matter of time before a more sophisticated solution will be available.

Chapter 5

Dozer Revisited

After Dozer's initial release we investigated the performance results and considered the lessons we had learned from this first release. In general the system was working extremely well and produced very good results. Nonetheless we saw several points where we were not completely satisfied with the system and expected to be able to improve the performance by tweaking specific subsystems.

The most important point where we saw room for improvement was clock drift handling. The initial Dozer release only used worst-case guard times to cope with clock drift and although the time spent in idle listening per transmission was only in the order of one millisecond it was already a significant factor of the total energy consumption. Similarly, we reinvestigated our management of potential parents for a node in the tree and the buffering strategy. Furthermore, we analyzed the behavior of the system in case of multiple sinks and also ran outdoor tests. In this section we give a summary of what we changed as compared to the initial release and what impact these changes had on the system.

5.1 Impact of Time Synchronization on Energy Consumption

The basis for all time-related operations on a sensor node is an oscillating crystal. The rate at which a node operates is defined by the oscillation frequency of the employed quartz crystal. As no two crystals physically oscillate in total synchrony clocks on different nodes in a sensor network never tick at the same rate. Consequently, algorithms to compensate for these

physical differences become necessary. In this chapter we discuss the impact of time synchronization on the energy consumption in a sensor network.

5.1.1 Fundamentals

Clocks on different devices in a sensor network tick at different rates. The reason for this difference is found in the slightly changing physical attributes of the oscillating crystals used to clock the CPU and timers of each device. Every crystal has a slightly different oscillation rate and is usually sold with a guaranteed bound on how much it differs from “real-time” at room temperature. Common low-cost crystals as we find them on current sensor nodes come with a drift of up to 50 parts per million (ppm). At room temperature a crystal which should oscillate at 1 MHz will be off by at max plus or minus 50 Hz; or if we look at the impact on clock values this corresponds to a maximal drift of 50 microseconds per second. This drift is symmetric and clocks can be faster or slower than their nominal rate. Consequently, the relative clock drift between any two nodes in the network may be up to 100 microseconds per second. Unfortunately, this bound is only valid at 25℃ temperature since clock precision degrades if nodes face changing temperatures. The change of clock rate versus temperature is usually modeled as a quadratic function

$$f = f_0 * [1 - alpha * (T - T_0)^2]$$

f is the measured clock rate at temperature T and f_0 the clock rate at temperature T_0. As the quadratic term $(T - T_0)^2$ shows clocks get slower independently of whether the device is heated up or cooled down. A worst case scenario in terms of clock rate offset is therefore constructed by keeping one node at 25℃ and the other node at the upper or lower bound of the range of acceptable operating temperatures. Under such extreme conditions drifts of up to 300 ppm may occur. While these extreme cases will not happen regularly in real-world deployments, temperature differences of up to 40 degrees can easily be found. A common scenario where such differences may arise is one node placed in an air conditioned building and the other node in front of the window in the plain sun. In such a setup relative drifts of 200-300 ppm have to be handled. Furthermore, as drift is incurred by temperature differences it is not stable but fluctuates. The day-night cycle or sudden changes of weather are two examples for sources leading to changing temperatures and thus changing clock offsets.

5.1.2 Clock Synchronization

Synchronization of clocks on different devices is a problem which can be tackled in various ways, all offering individual advantages and drawbacks. In

general, a one shot solution synchronizing clocks once is unfeasible due to dynamic change of clock rates as described in the last section. Thus, algorithms which continuously synchronize clocks or at least do so periodically are required. The most primitive approach of synchronizing devices is to periodically update the current clock value on the devices. The problem with this approach of clock synchronization is that directly after the synchronization event the clocks start to drift apart again, as their different clock rates are not taken into account. A more sophisticated approach to time synchronization is therefore not only to work on the value of the time register but also to modify the clock rate. This is more difficult to do as the clock rate is defined by the physical attributes of the oscillating crystal and thus lies beyond the control of the software. It is therefore necessary to add a further level of abstraction by introducing a "virtual clock". This clock maps the value of the physical clock to a virtual time which is then used by the application. Depending on the algorithm used, an arbitrary complex formula may be applied to compute the virtual time including the clock value and rate. Recent work has shown that achieving network wide synchronization is difficult [25] both from a theoretical [11, 26] as from a practical [27] point of view and that also prominent protocols such as TPSN [14] suffer from inherent scalability problems.

5.1.3 Time Synchronization and Communication Protocols

Many communication protocols heavily depend on clock synchronization. Especially TDMA-based and similarly scheduled communication systems are vulnerable towards clock drift. If a pre-computed rendezvous time between two communication partners is not met correctly by both involved devices, lost transmissions and unnecessary packet collisions occur. The degree of synchrony decides on the energy efficiency such protocols can achieve. In case of perfectly synchronized clocks and totally stable clock rates nodes might sleep for 100% of the time while they are not actively participating in a communication process. They could only turn on their radio right on time for the transmission and then directly go back to sleep. With less than perfect synchronization and changing clock rates, additional guard times are required to protect against transmission problems. The receiver has to wake up slightly early to make sure to be in receiving mode when the expected transmission starts. How much "slightly early" is depends on two factors: First, the precision of the clock synchronization and second, the maximal possible drift change since the last synchronization. The precision aspect thereby defines how exact a synchronization algorithm manages to judge the

different clock rates and values. In other words, it is the estimation error in a scenario of totally stable clock rates as it may be witnessed if all nodes are kept at exactly the same temperature all the time. All clock synchronization algorithms aim at bringing this error to zero.

5.2 Clock Drift Compensation in Dozer

The initial implementation of Dozer's network stack dealt with clock imprecision using worst-case guard times. For each message reception the radio was turned on in time to guarantee a successful transmission at a relative drift of up to 100 ppm. With a beacon interval and thus synchronization period of 30 seconds these 100 ppm result in a guard time of 4.5 milliseconds. Compared to the 4.7 milliseconds required for the actual transmission of a data message this is a significant overhead. Since the majority of all transmissions between two nodes do not require worst-case guard times a more sophisticated drift compensation mechanism has the potential to preserve a large amount of energy.

An appropriate drift compensation component for Dozer has to feature several properties. First, it should not demand additional message exchanges. Considering Dozer's low rate of maintenance traffic the cost of extra synchronization messages would not only nullify their positive effect but even increase total energy consumption. Second, due to the inherent limitations of current sensor network hardware in terms of memory and computation power complex calculations and data structures must not be relied on. After a close analysis of our requirements we came to the conclusion that we did not need a global synchronization of the network. Similarly to the two independent communication schedules a node maintains, our synchronization requirements are only between a parent and its direct children. Such a local synchronization is much simpler to realize than a global one and we were therefore able to develop a clock synchronization mechanism solely recycling information produced by the tree maintenance component.

For the initial release of Dozer we employed the common approach of having the receiver compensate the clock skew. This approach is reasonable when using fixed guard times but it suffers from a scalability problem if dynamic drift compensation is applied. Each node in Dozer's data gathering tree can have a large number of children and would thus be obliged to compute and maintain an individual drift prediction for each of them. Dozer avoids this issue by exploiting the tree structure of the network. It burdens the child with the task of drift compensation for all communication with its parent—independent of whether it is sender or receiver. Hence, each node in the network only has to handle the drift compensation for one bidirectional

connection.

With the periodic beacon transmission a message exchange is available which can be used to synchronize a child with its parent. When a child connects to a new parent it is unable to predict their relative drift. Consequently, it uses a worst-case guard time of 200ppm. On reception of the next parent beacon it time-stamps the message with its system time. From the information contained in the beacon the child is able to compute how much time has passed between the last two consecutive beacon transmissions according to its parent's clock. Based on this value and its own local timestamps the child derives the current relative drift between the two nodes. For the next beacon reception the child incorporates its current drift. This process is repeated on each beacon reception in order to maintain an up-to-date drift prediction.

Drift estimation is only half of what makes a drift compensation system. It returns a point in time when the next beacon message is expected to arrive. However, this is only an expected value and a drift compensation system also needs to know how much the actual arrival time may vary from this prediction. This knowledge is essential as it is needed to determine how much in advance and for how long the radio needs to be turned on in order to maximize packet reception while minimizing energy consumption; in other words the system has to decide what guard times to use. In a perfectly stable system this guard time could be set to zero but in real-world scenarios drift changes over time. Even an optimal drift estimation system fails if the environmental conditions abruptly change in between two synchronization cycles.

Dozer deals with this problem by starting out with a large guard time which rapidly shrinks towards a defined minimum. As described before a newly connected node uses a worst-case guard time for the first beacon reception. After receiving the next beacon the difference between the predicted and the actual reception time, denoted as estimation error, is used as guard time for the next communication. As a consequence, the employed guard times rapidly converge towards zero. Since the system still has to cope with sudden changes in the environment we limit the minimum guard time to the maximum expected drift change within one beacon interval. This value is hardware dependent and was empirically determined for the TinyNode platform. To do so, a two-node setup consisting of one sink and one child node was employed. Initially both devices were kept at room temperature for approximately 15 minutes. After this period, the child was placed in a freezer and cooled down to -20℃. As shown in Figure 5.1 the temperature difference between the two nodes grew rapidly and the estimated relative drift increased by 60 ppm. Nevertheless, the estimation error indicating how much the child was mistaken in its prediction of the next beacon arrival time never exceeded

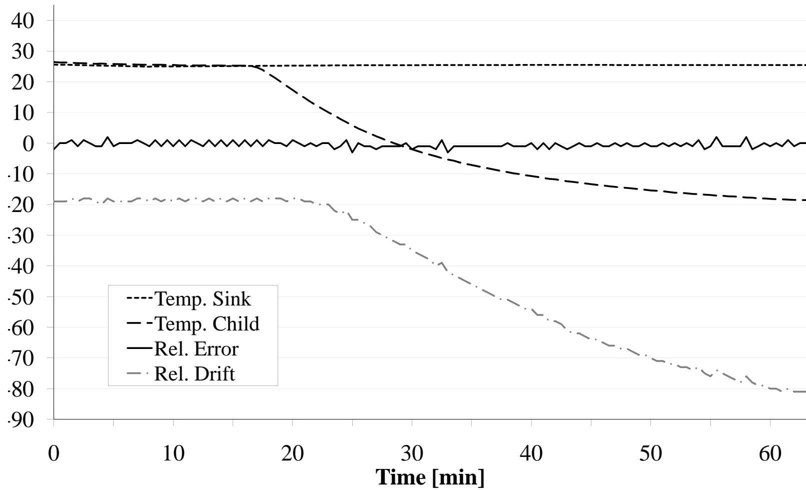

Figure 5.1: Drift estimation and error of a two node setup with one node at room temperature and one in a freezer.

3 jiffies[1] with an average error of 0.88 jiffies. In this experiment the nodes were not confronted with the maximum or minimum service temperature and thus the relative drift between them did not exceed 81 ppm. However, the speed at which they changed their temperature gives a good estimation of what is to be expected in common outdoor deployments. Based on the maximal measured estimation error of 3 jiffies we decided to set the minimum guard time to 20 jiffies what corresponds to 0.6 ms. This conservative choice enabled us to run Dozer in both, indoor and outdoor deployments with the same settings.

In practice it still has to be expected that the system sporadically fails to receive transmissions. In this case the guard time is immediately set back to its initial worst-case value. Consequently, a trade-off between the size of the minimum guard time and the number of failed transmissions has to be made. For networks operating at more or less stable temperatures such as indoor deployments, an aggressive minimum guard time results in best performance. On the other hand, networks which have to deal with frequent temperature changes such as most outdoor deployments benefit from a more conservative setting.

[1] 1 jiffy = 1 clock tick = 1/32768 s

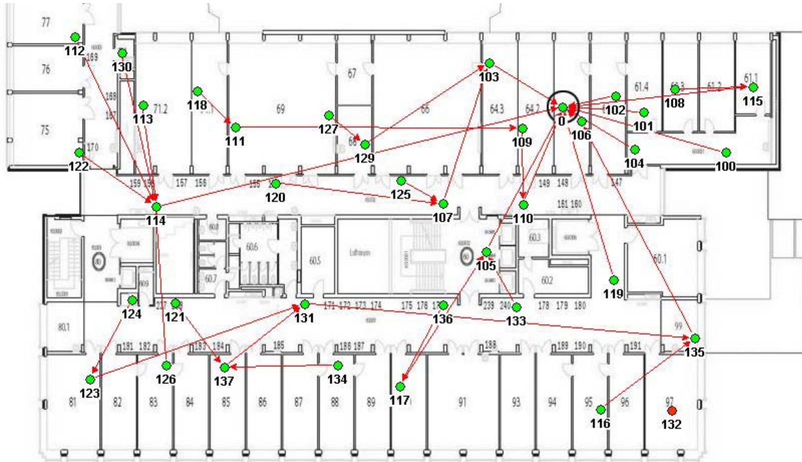

Figure 5.2: Indoor deployment of 39 nodes. Node 0 (upper-right corner) acts as data sink.

5.3 Experimental Evaluation

In this section we conduct an in-depth analysis of Dozer's performance in indoor and outdoor scenarios highlighting different aspects of the system. The experiments were once again run on the TinyNode 584 [17] sensor nodes operating on TinyOS 1.x.

5.3.1 Office Floor Experiment

As a first step we reproduced the office floor setup which served as a benchmark for the initial implementation to assess the influence of the introduced enhancements and modifications on the power consumption of individual nodes. The setup consisted of 38 sensing nodes and one data sink (node 0) placed across a floor of our office building as depicted in Figure 5.2. Node 132 failed shortly after the initial deployment and did not recover.[2] As a consequence this node was excluded from all further calculations.

Dozer was configured analogously to the original experiment with a beacon interval time of 30 seconds and a data sampling rate of one measurement every two minutes. The drift compensation system was set up to allow for

[2]Later investigations unveiled a defective power source to be the root of the problem.

up to 200 ppm of relative drift and a maximal drift change of 20 ppm per beacon interval as defined in Section 5.2.

In this office floor setup the initial implementation—compensating up to 100 ppm of relative drift—produced an average radio duty cycle of 0.167% for the sensing nodes. A repetition of the same experiment using the improved Dozer protocol resulted in a mean duty cycle of 0.128% which corresponds to an improvement of more than 23%. This significant reduction in power consumption is achieved by different system optimizations. The lion's share of the gained radio sleep time stems from the drift compensation component. Looking at the minimum duty cycle of a leaf node a reduction from 0.07% with the old implementation to 0.057% in the new release is found. This observed performance gain can be accredited to the drift compensation component as no other modifications influence this specific value. Consequently, this component accounts for approximately 18% reduced radio uptime. The remaining 5% result from a change in the potential parents mechanism. As described in Section 2.1 each node maintains a list of neighboring nodes which may serve as a parent in case of a problem with the current connection. In the initial release this list used to be refreshed regularly in order to maintain up-to-date information about known neighbors. Now, the update procedure is disabled. The drawback of this decision are increased setup costs when a new parent is required as it is harder to time a rendezvous with neighboring nodes based on old information. Yet, these costs are more than compensated by the amount of energy saved by not communicating regularly with all nodes on the potential parents list. A detailed overview of the average duty cycles for all deployed sensing nodes is given in Figure 5.3.

In the following we investigate the overhead incurred by clock imprecision and guard times at a leaf node of the data gathering tree. We assume an ideal system transmitting the same number of messages as Dozer in the given configuration but with perfectly synchronized clocks. A leaf needs to exchange 4.5 messages per minute. A message including its preamble is 44 bytes long, leading to a transmission time of 4.7 ms at 75 kbps. The radio switching times from sleep to transmit and back add up to 2 ms. These parameters result in an ideal duty cycle of 0.0525%. Thus our initial system, exhibiting a duty cycle of 0.07%, suffers from an overhead of 33%. In comparison, the new system with a duty cycle of 0.057% reduces this overhead to 8.57%.

Another interesting observation of the indoor experiments is the lack of any message losses; all generated data samples reached the sink. In contrast, our initial experiments produced an average loss of 1.2% in the same setting. This improvement originates from a changed buffering policy. Dozer uses link layer acknowledgments for all data traffic. Therefore, no messages are lost in the course of their transmission. However, due to its limited amount of memory a node can only buffer a finite number of messages. The buffering

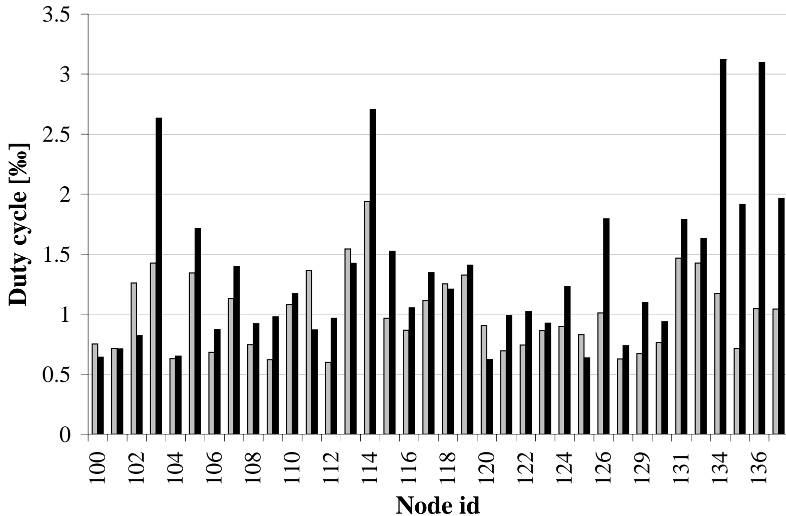

Figure 5.3: Average radio duty cycles in per mill of all nodes with one (black) and two (grey) sinks, respectively, in the office floor experiments.

strategy applied in the initial release allowed only one message from each originator in a node's forwarding queue. If more than one packet from the same originator met in a node the older message was discarded. Now, Dozer was configured to store an arbitrary number of messages from each source as long as the required buffer space was available. Since no network interruptions of more than a couple of minutes occurred during the one week of the experiment no buffer overflows happened and perfect message yield was achieved.

Note that in case of hardware failures the current Dozer implementation would lose the messages buffered in the failing node. For the majority of application scenarios this loss is negligible. However, if not a single message must be lost mirroring the message queue in the node's permanent storage would guarantee a possibility to recover the corresponding data. As EEPROM access is expensive in terms of energy consumption (same order of magnitude as a radio message) this solution creates additional overhead and should therefore only be used when absolutely needed.

5.3.2 Multiple Sinks

A challenge every multi-hop data gathering system faces is increased load on nodes close to the sink. Serving as a relay for most data messages these nodes are forced to handle higher traffic rates than other nodes in the network. As a consequence their duty cycles increase and their batteries deplete at a faster rate. Furthermore, due to their strategically important position in the network their failure usually also marks the end of the whole network as the remaining nodes cannot reach the sink any longer. Diminishing the increased load at nodes close to the sink is accomplished by reducing the amount of traffic actually reaching the sink. There are two feasible approaches to achieve this goal. First, in-network processing and aggregation may be applied to condense the forwarded information. Depending on the nature of the sampled data this optimization may result in a significant improvement. Unfortunately, in many scenarios neither in-network processing nor aggregation can be applied since the full sampled information of each node is required. Under these conditions the problem can only be countered by the sensible deployment of additional sinks to spread the load on more nodes.

Dozer is designed to handle dynamic addition and removal of sinks. If more than one sink is available a separate data gathering tree for each of them is constructed without any additional overhead. All nodes in the network select a parent with minimal distance to any of these sinks. The different trees are thereby not labeled and thus nodes do not know to which of them they belong. As a consequence, nodes—and thus whole sub-trees—may freely switch from one data gathering tree to another without even noticing. The advantage of this system lies in its flexibility towards load and interference. External interference may cut off parts of the network or enforce long detours to bypass error-prone areas. In such cases the presence of additional sinks may help avoiding data loss and generally reduces maximum tree depths. Hence, not only load at nodes close to the sink is reduced but also the global average duty cycle is improved.

Based on the network introduced in Section 3.3 we ran an additional experiment with a second sink. The two data sinks were placed at opposite sides of the building to provide a reasonable load balancing. As shown in Figure 5.4 the second sink (node 1) had the desired effect in that two nearly equal sized trees were constructed. Nodes in the central area of the building regularly switched between parents in either tree since their hop count towards both sinks was balanced. With the decrease in tree depth the maximum observed load measurably dropped resulting in a global average duty cycle of 0.093%—compared to 0.128% with one sink. Figure 5.3 depicts the average duty cycle for each node in the network with one and two sinks,

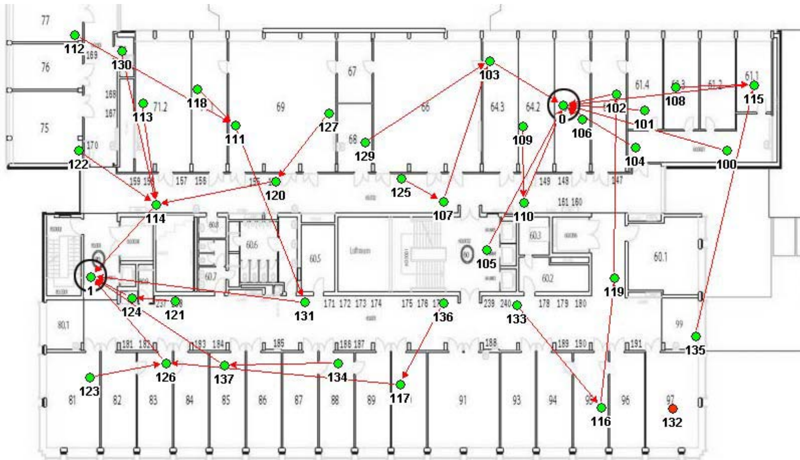

Figure 5.4: Indoor deployment of 39 nodes. Node 0 (upper-right corner) and Node 1 (middle left) act as data sinks.

respectively. The majority of all nodes benefited from the presence of the second sink and the reasons for the individual improvements are manyfold. For example, nodes 134 and 136 were affiliated to the tree of node 1 avoiding a link through the center of the building. In the single-sink experiment both of them had to rely on this link which apparently suffered from high packet loss. Other nodes such as node 107 had to service a lighter subtree and therefore spent less energy forwarding messages. Following the same line of argument it can also be explained why some few nodes such as node 102 suffered from an increased duty cycle in the two sink setup.

5.3.3 Outdoor Experiments

To get a notion of Dozer's behavior in outdoor scenarios we ran an experiment consisting of 30 nodes for a period of 3 days in a picturesque suburb of Zurich, Switzerland. The site was located in a quiet residential district on a hillside bordering a small forest (see Figure 5.6). While most of the nodes were deployed in the open, three nodes (100, 105, and 116) plus the data sink (node 0) were placed indoors. The test application running on top of Dozer was designed to sense the intensity of the ambient light as well as the temperature. The weather was quite unsettled during the operation of the network ranging from a thunderstorm with heavy rain on the first day, to

hot sunny weather on day three (see Figure 5.8 for a detailed history of the nodes' temperatures). To be able to cope with these conditions appropriate housing for the sensor nodes was required to protect them from undesired operating conditions without hampering their sensing abilities. The design of sophisticated housing solutions is non-trivial and went beyond the scope of our tests. We therefore deployed the TinyNodes in simple waterproof boxes with a transparent top completely sealing in the node and its sensors (c.f. Figure 5.5). The housing of node 125 turned out to be leaky and water ingress resulted in unpredictable node behavior varying from normal operation to total breakdown for more than a day. Consequently, this node was excluded from all further observations.

From an analytical point of view we anticipated Dozer to perform equally well in this setting as in the indoor experiments shown in Section 3.3. With a measured average duty cycle of 0.172% this expectation was met to a certain extent. However, the measurably increased radio uptime calls for a closer investigation. In general, all nodes suffer from a slightly increased duty cycle. This can be explained by the more dynamic nature of the outdoor environment. Changing weather conditions influenced the characteristics of the wireless channel and enforced a higher degree of adaptation. This is for example reflected in a decreased link lifetime. On average a link remained stable for 11.5 hours in the indoor testbed. In the outdoor setting this value dropped to 8.7 hours.

Furthermore, the data gathering tree converged to a relatively deep but stable topology. As a consequence, many nodes had to serve as relays. All nodes exhibiting duty cycles of more than 0.2% (c.f. Figure 5.7) were affected by this phenomenon. In particular, node 102 and node 115 were forced to forward all traffic for the whole duration of the experiment since they represented the only stable bridge between the sink and the rest of the network. These two nodes are an example of the problem discussed in Section 5.3.2. Consuming significantly more energy than all other nodes in the network they are the first to fail due to depleted batteries. As the other nodes will then no longer be able to reach the sink, the network becomes useless as soon as these two nodes fail.

Node 137 exhibits a unexceptional behavior. Despite being a leaf its duty cycle exceeded 0.2%. From a close inspection we found that on this node one important energy saving mechanism of Dozer was annulled. In order to associate a child with a new parent a two way handshake is executed in a contention window following the beacon message of the parent. Since these connection changes are rare the window has to be explicitly activated by the child. For this purpose the child transmits a busy tone right after the reception of its parent's beacon message. Simultaneously, the parent conducts a RSSI sniff to decide whether to stay awake or go back to sleep.

Figure 5.5: Location of the outdoor deployment and an encased TinyNode.

On node 137 this sniff failed, always reporting a busy channel[3]. Hence, the node remained listening for 20 ms after sending its beacon resulting in an additional duty cycle of 1‰. Further investigations revealed that also the other two nodes located on the same terrace (node 102 and 115) suffered from this problem. Due to the nature of the environment we were unable to identify the source of interference.

As in the indoor setup, Dozer managed to reliably transfer all data to the sink in normal operation. However, one anomaly was registered at node 101 in the early afternoon on the third day of the experiment; the node spontaneously rebooted twice. These restarts were a direct result of the warm weather conditions. As depicted in Figure 5.8, the reported temperatures of all outdoor nodes were beyond 60°C at that time. It is to be noted that these readings do not reflect the ambient temperate inside the box but represent the temperature on the node's surface. This value is significantly higher since exposure to direct sunlight heated up the hardware. The last received samples of the failing node before its reboots indicated temperatures beyond

[3] A RSSI sniff captures energy within the whole available frequency band of the radio and not only in the currently used channel. Consequently, even activity in adjacent frequencies may lead to false positives indicating that the channel is in use although it is not.

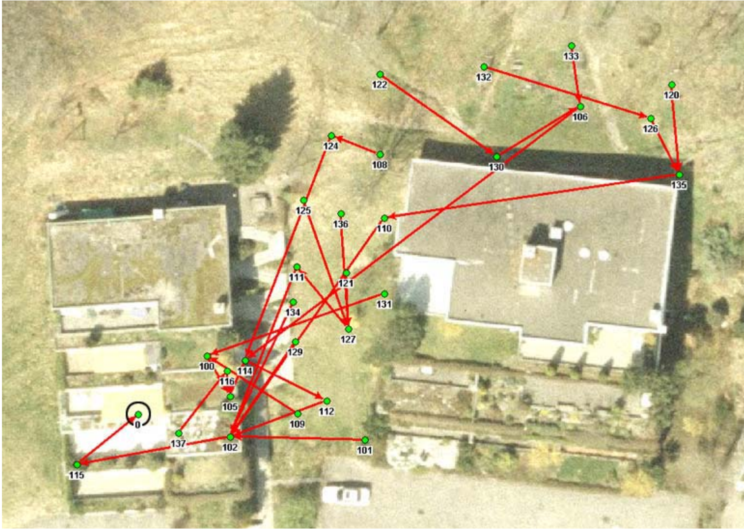

Figure 5.6: Outdoor deployment of 30 nodes including a snapshot of the data gathering tree. node 0 (lower-left corner) acts as data sink.

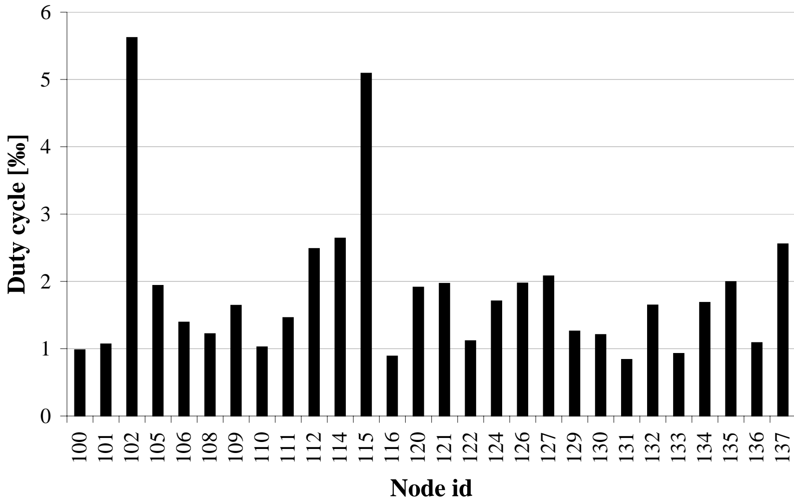

Figure 5.7: Average radio duty cycle of all nodes in the outdoor deployment.

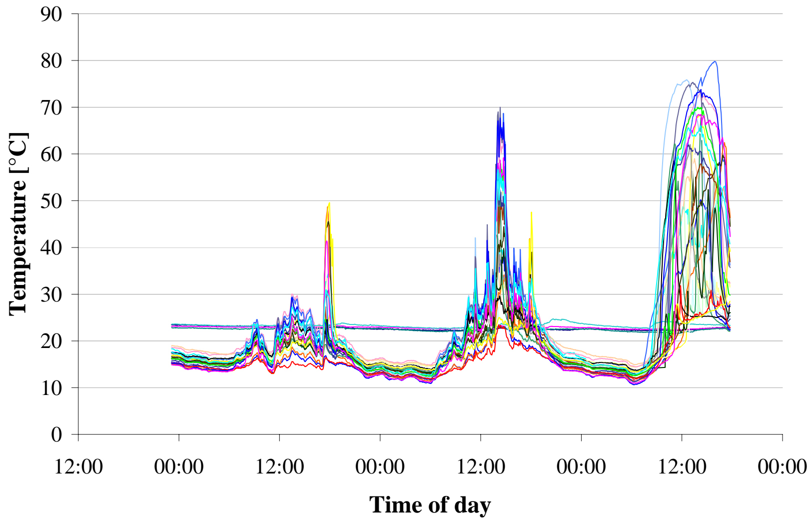

Figure 5.8: Temperature of all nodes over three days. The steady curves at 23℃ originate from indoor nodes. Day 1: thunderstorm; Day 2: unsettled; Day 3: sunny.

75℃. That is, the node was operating close to its maximum approved service temperature. We assume that at the moment of the actual reboots the node faced even higher temperatures leading to unpredictable behavior and eventually a system restart. Node 101 was a leaf in the tree and thus its temporal malfunction had no effect on the other nodes in the network.

Excepts for nodes 102 and 115, all nodes in the network ran at less than 0.3% duty cycle. Applying the values from Table 3.1 a 0.3% duty cycle results in 0.136 mW power consumption. Assuming a power source with 2000 mAh at 3 volt the first node failures occur after more than 5 years. The average node lifetime for this experiment is above 8 years.

Chapter 6

Dozer in the Wild

In this chapter we give an overview of a real world project using the Dozer communication stack and an evaluation of Dozers performance under harsh conditions. The project is called “PermaSense” and is a joint venture between the Computer Engineering group at ETH Zurich and the Glaciology, Geomorphodynamics & Geochronology group at University of Zurich. We are not directly participating in this project but have close relations to the Computer Engineering Group and are thus closely following the project.

6.1 About PermaSense

The goal of PermaSense is to learn more about the impact of climatic changes on rock fall in permafrost areas. To gather this data a dense wireless sensor network grid is deployed featuring custom sensors to measure changes in the rock and ice. Such a study has only become possible with wireless sensor network technology as so far there were no easy to deploy geo-monitoring systems that are low-cost, cheap in maintenance, and easy to reconfigure also after deployment. Beyond the current scientific measurements the PermaSense system may also be used to permanently monitor large natural hazard areas and in combination with a warning system help to protect human lives.

Currently, the PermaSense project operates two deployments, one on the Jungfraujoch (see Figure 6.2) and one on the Matterhorn. Both of these deployments have been in operation for more than a year. The majority of maintenance is remote controlled using a combination of wireless technologies and telephony. On-site work (also see Figure 6.1) involves a trip on the mountains and is thus tedious and expensive. On the Jungfraujoch the “Jungfraujoch railway” can be used to get close to the deployment. This

Figure 6.1: Difficult deployment of a sensor node on the Jungfraujoch. [source: http://www.permasense.ch/]

is a significant advantage over the Matterhorn deployment where extensive climbing or a helicopter flight is necessary to access the nodes. Therefore, the Jungfraujoch was chosen as a first deployment site.

6.2 Dozer to PermaDozer

Unlike in our own experiments the PermaSense project is no testbed but a wireless sensor network deployment with the goal of collecting meaningful information which is to be used by Geologists in their studies. This is also what makes this project so interesting, as the interaction of different components such as the complex involved sensors have a strong impact on the operation of the communication stack. In this section we discuss how Dozer is integrated in the PermaSense project and what changes were made to turn Dozer into PermaDozer running on the PermaSense nodes.

In PermaSense the communication parameters are set analogously to our own setups including a beacon interval of 30 seconds and a data sample rate

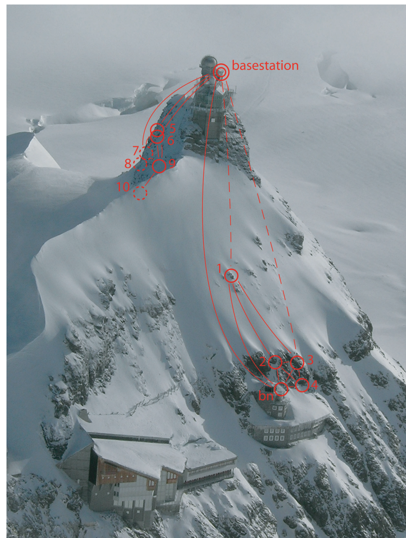

Figure 6.2: Current PermaSense deployment on the Jungfraujoch. [source: http://www.permasense.ch/]

of one every two minutes. A first significant change to the protocol became necessary as the time to sample one of the employed sensors takes up to five seconds and thus exceeds Dozer's default frame length during which the application is free to block system resources without interfering with the communication stack. This is however a minor problem as Dozer can easily be modified to give the application larger time frames during which no critical communication is scheduled. This modification results in a slight reduction of message throughput as less time is reserved for communication but at the given data sampling rates this results in no measurable additional delays. Interestingly, although designed to protect the communication stack from interference of the sensors and application, this separation also turned out to be a vital requirement to generate precise sensor readings. The PermaSense team has shown that there is a significant interference within the sensor node if all components are activated simultaneously. That is, the RF components can distort the readings of the employed analogous sensors. Therefore, Dozer's coarse grained separation of communication and sensing intervals turned out not only to help prevent message loss but also to increase the quality of the sampled data.

Another challenge faced by the PermaSense deployments is the situation where one or more sensor nodes are completely snowed in and cannot communicate with other sensor nodes any longer. This condition may remain for months at a time and the sensing devices thus need a way to store their readings "offline" as the queue in RAM is insufficient to buffer all readings. In PermaSense this problem was solved by adding an SD card to each sensor node which serves as a storage medium for all captured sensor readings. Once a node regains connectivity it dumps its buffered data using the Dozer communication stack. Although Dozer was not designed for such sudden bursts in data, the protocol can handle the load. Dumping the readings stored over a month of offline operation will take time as only approximately 10 messages can be forwarded per communication round but the default acknowledgement system suffices to make sure no messages are lost on the way and eventually all data is recovered at the base station. Using a simple time-stamping mechanism the readings can also be sorted chronologically and thus as soon as communication is possible again the readings of the snowed in nodes gradually appear in the database. Furthermore, the SD storage can also be used to verify data integrity after a year or two of operation. By comparing the data samples gathered over the wireless connection with the data backed up on the SD cards it can be proven, that no data corruption is incurred on the wireless medium. This proof is important to convince other involved parties such as the geologists working with the gathered data, that wireless sensor network technology is as reliable as traditional measuring systems.

Another interesting challenge is the deployment of the nodes. PermaSense

uses practically the same deployment/disconnection protocol as we used in our own testbeds. That is, a node periodically sniffs the channel for any messages and if it finds activity starts listening for a beacon interval to check for beacon messages. As this system can be set to spend similar power as a connected node it is well suited for the operation of a network which may be physically disconnected for months at a time. However, the drawback is the slow integration of sleeping nodes in the network. It is no problem if a node that was offline for a month takes a day before detecting that the connection to other nodes is once again possible. However, during the deployment phase this slow reaction is a problem. It may take hours before a newly deployed node reports its first reading to the database and the correct operation of the sensors can be verified if it was in sleep mode during the deployment. Additionally, the meaningless sensor readings generated between the time when a sensor node is powered up and the time it is deployed (which may be several days) should be removed from the memory card to prevent logging bogus data. The simplest way to achieve these goals is to reset the nodes and wipe the SD storage on deployment. As conditions on the mountains are often damp, opening the water sealed boxes housing the electronics is no option. PermaSense nodes have therefore two reed contacts and a buzzer allowing a non-intrusive way to manually reset the sensor node and wipe the flash memory on deployment.

These are the main modifications the PermaSense team has made to the Dozer system to optimize it for use in the alpine deployments. While they may seem minor from a conceptual point of view their importance for the successful operation of the wireless sensor network must not be underestimated. From the PermaSense project we have learned that the integration of multiple components in a sensor node requires much more work than simply plugging them together. Only if this engineering step is executed with care the complete system may operate at maximum performance.

6.3 Performance

Energy efficiency is vital for PermaSense as an unattended network life time of three years is planned. During this time nodes have to operate on a non-rechargeable battery capable of coping with the extreme ambient temperatures ranging from -40 to +65 ℃. In [1] the PermaSense team gives a detailed analysis of the power consumption of their entire system. Comparing the performance of Dozer in PermaSense to our own measurements is not trivial as in PermaSense different constraints apply. For example a single measurement results in five TinyOS messages to be sent by the node as there is too much payload for a single message. Furthermore, in Per-

Operating Mode Characterization	**[mA]**
Sleep	0.026
DAQ active	2.086
Dozer RX idle	13.64
Dozer RX	14.2
Dozer TX	54.6
Measured Average Values	**[mA]**
DAQ only (2 min)	0.110
Dozer only (30 sec / 2 min)	0.072
PermaDozer total (30 sec / 2 min)	**0.148**

Table 6.1: Power consumption in PermaSense for different operating modes at 3.6V

maSense the transmission power of the nodes is higher than in our testbeds. In our experiments we used a transmission power of 40.25 mW whereas in PermaSense 196.6 mW are used. Similarly, due to the much more complex hardware employed in PermaSense the sleep power drain of their nodes is higher than for a vanilla Tinynode and has risen from 0.006 mA to 0.026 mA. Nonetheless we will do a rough analysis of the achieved duty cycles in PermaSense based on the available numbers. The PermaSense team has conducted power measurements for the individual components of their system as listed in Table 6.1.

To compute the radio duty cycle of a node based on these values we make the following assumptions. In every minute a node sends two beacon messages and receives two beacons from its parent. Furthermore, it transmits 0.5 data samples which corresponds to 2.5 messages. That is, 4.5 messages are sent and 2 messages are received. Using these numbers and the values from Table 6.1 to compute the mean power consumption, a node draws on average 42.17 mA per transmission. Combined with the sleep power consumption of 0.026 mA and the total average power consumption of 0.072 mA we can compute an estimated radio duty cycle of 0.11%. As in PermaSense 5 data packets are transmitted per sensor sampling and not one as in our testbeds, the message count per minute raises from 4.5 to 6.5, what corresponds to an increase by 44%. If we factor in these additional messages the measured duty cycle of 0.11% corresponds to a duty cycle of 0.076% in our experiments and is therefore very close to our own findings. Also the plot over time for the power consumption of three nodes as given in [1] and depicted in Figure 6.3 matches exactly our expectations and shows the proper operation of the data acquisitions (DAQ) and communication stack.

Another observation of the PermaSense project is that communication is

no longer the main source of power consumption in their system. A node draws on average 0.148 mA. This power consumption can be split into three main contributing factors: Sleep mode plus overhead, data acquisition, and Dozer. Sleep mode thereby accounts for 23%, data acquisition for 51.4%, and Dozer for 25.7%. Simplified, one may say that the PermaSense system buys "real-time" time data aggregation at the price of 25% of their total energy budget.

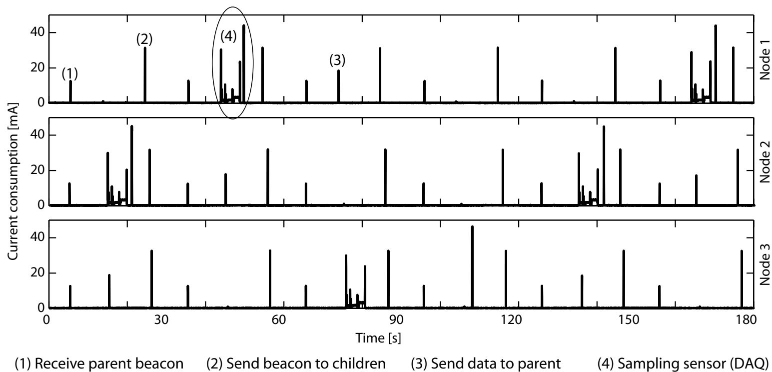

Figure 6.3: Power analysis for three nodes without children in PermaSense as given in [1]

6.4 Conclusions

Our Dozer code is not available to the public due to the commercial rights by Shockfish. We are therefore very happy with the special agreement allowing our colleagues from the PermaSense project to use Dozer for their deployments. For the first time external developers had the chance to use our protocols and thus to proof the performance claims we made for our system. With the extreme application scenario in the Swiss Alps it is also a perfect showcase for the viability of our system under harsh conditions. We are very pleased with Dozer's performance in PermaSense and that no major changes to the system were necessary to use it in this project.

We are especially proud of the finding that in PermaSense the communication stack no longer represents the main power consumer of the system. This finding is in contrast to the common belief that the communication system is the most expensive component in terms of energy in any real-world sensor network [43, 44]. The PermaSense team concludes that the reason

for this finding is the previous lack of power analysis in integrated systems as compared to individual components. We agree that this different form of power analysis has an impact on the precise distribution of the energy budget. However, the impact is not large enough to lead to such a significant shift in the distribution of the power consumption. PermaSense, although operating under extreme conditions, falls precisely in the category of data gathering applications Dozer is designed for. Therefore, PermaSense benefits from optimal performance of our system and we know of no other communication stack for sensor networks which could achieve comparably good results in this scenario.

Chapter 7

Related Work and Comparison

Before we come to the actual related work we make a short excurse to the problems faced when comparing different wireless sensor network applications. One of the basic rules in science is to compare your results with existing work. Obviously this requirement helps appraising the impact of new ideas and may also show hidden strengths and weaknesses. However, when talking about communication protocols it is far from trivial to design fair tests to compare different protocols with each other. Especially when looking at energy efficient MAC-layers and routing protocols it quickly becomes clear that it is rather trivial to come up with test cases where one protocol will perform exceptionally badly whereas another one may shine. For all algorithms exist traffic patterns where a minimal overhead is produced and others where the vast majority of energy is spent in vain. Developers are aware of this and may be tempted to benchmark an algorithm on a scenario where the own work has the edge on its competitors. This is not even necessarily a bad thing as—while this is also true in almost all other areas of life—especially in sensor networks custom solutions for specific applications always perform best. Hence, it is natural to use a test case for which the new algorithm was designed. However, one has to keep in mind that results of such tests usually do not translate well to other conditions. Also the kind and quality of the implementation has a significant impact on the outcome. Pure simulation results are often far off of what we witness in real-world deployments. Communication models are either too simplified or too harsh, timing problems (e.g. due to changes in temperature) are often neglected, traffic patterns are artificially chosen, and interference follows the strict rules of a model. As a

consequence, when dealing with numbers deduced from simulations the limitations of the simulation need to be kept in mind and thus not all results may be taken at face value.

Unfortunately, also measurements on testbeds consisting of hardware nodes can differ significantly. The topology of the network, the number of deployed nodes, as well as the external conditions (e.g. air conditioned or outdoor deployment) have a major impact on the achieved results. In addition the employed hardware is of importance. If the radio has good switching times between off and RX/TX and can cheaply conduct an RSSI sniff a protocol based on low-power listening (cf. Section 7.1) will perform much better than on hardware with a slow startup and expensive radio sniffs. Also the quality of the implementation has to be considered as a proof of concept implementation will perform significantly worse than a fully debuged version of the same system. Unfortunately, for many of the more sophisticated protocols no optimized implementations are available, making direct comparisons on a testbed very difficult.

An even bigger problem is finding good metrics to compare protocols. For example, how to measure energy efficiency? Energy efficiency is the holy grail of wireless sensor network development and finding the most efficient protocols for a deployment may make the difference between a successful project and a total failure. Still, there is no standardized approach for measuring the efficiency of a protocol. As described before, due to the varying demands of different applications no single benchmark can be fair for all purposes. However, many of the existing communication stacks can be clustered based on the employed basic schemes. For example there are numerous LPL-based [9, 42] protocols and TDMA-based [20, 45] approaches. Nonetheless, there is no standardized benchmark setting for either of these classes. As a result authors choose different parameters when testing their systems and represent their findings in various formats. Sometimes energy consumption is given in milliwatt other times the radio duty cycle notation is used. The former has the drawback that it is extremely hardware dependent and thus results cannot be compared to tests conducted on another platform. The latter translates better to other hardware although radio switching times still have a strong impact on the measurements. Both approaches suffer from the problem that the data rate and traffic pattern of the test application running on top of the communication stack has a strong influence on the results. A protocol may be very efficient if it has to send ten messages per second and node but still use a similar amount of energy if only one message per minute is sent; of course the opposite may also be true. Simply removing the payload completely and only benchmarking the consumption of a communication layer in idle state is no option either as the trivial algorithm not sending anything at all would be optimal in this setting.

To a certain extent the performance of a system under specific conditions can be extrapolated from values measured in a different setup. However, by doing so the advantages of real-world tests over simulation results get lost. The conclusion of this section is that we need standardized test cases for each different class of sensor network applications. The definition of what settings to use and how to standardize the setup of the testbed is far from trivial and will not only result in technical but also "political" discussions. Only if the whole community working in this area can agree on a set of test cases, more meaningful comparisons can be achieved.

In the absence of such standardized tests we have decided to use a mathematical approach to compare Dozer to its related work. The best possible numbers available for related algorithms were used to extrapolate their behavior under the same conditions we used to benchmark Dozer. The resulting numbers are definitely not precise but they suffice to give a strong indication that Dozer outperforms other protocols in the scenario it was designed for.

7.1 Comparison

In this section we compare the performance of Dozer with field-tested protocols deployed in environmental monitoring applications and recent related work published after Dozer.

7.1.1 LPL and Twinkle

First, low-power listening, or LPL in short, a strategy to condition contention-based medium access protocols to low-power requirements is evaluated against our system. This technique is included in B-MAC [42], the standard medium access protocol shipped with TinyOS. In LPL, the radio periodically probes the wireless channel for incoming packets. If no activity is detected the node returns to sleep mode. Otherwise, it remains on and the incoming packet is received. To ensure the receiver is listening a sender has to prefix its packets with a long preamble acting as an in-band busy tone.

Second, we compare Dozer to Twinkle [19]. Twinkle, a descendant of FPS [20], features some similarities to our system. It establishes a TDMA schedule where each node allots distinct time slots to all of its children thus allowing collision-free communication among them. The protocol thereby relies on a global coarse-grain time synchronization. Collisions between neighboring nodes which are neither siblings nor parent-child related in the tree are resolved using CSMA. Other protocols for low-power data gathering were proposed [28, 45, 64]. As they are evaluated by means of simulation or in single-hop networks only we refrain from comparing them with Dozer since

it is difficult to assess how well their results translate to a real-world deployment. An overview of these protocols and their pros and cons is given in Section 7.2.

Both LPL and Twinkle are evaluated on the mica2dot sensor node platform in [19]. Since no source code is available for the latter all measurements for the two protocols are directly taken from [19]. Similar to our indoor testbed discussed in Section 5.3 all tests were conducted on a 30 nodes indoor deployment. Different hardware platforms and testing environments hamper an in-depth comparison of the three protocols. Nevertheless, we gain a rough estimate of their performance in real-world deployments and their energy-saving capabilities.

Figure 7.1 depicts the average power consumption of the three considered contestants if one sample is generated every two minutes. Along the lines of [19] the energy drain for this sampling rate was extrapolated from existing measurements. For both LPL and Twinkle the shown values represent the power consumption of a leaf node. This lower-bounds the load of all nodes in the network since these nodes do not have to forward any external data. Low-power listening[1] consumes 2.83 mW on average. With Twinkle a leaf node experiences a power draw of 0.42 mW which is an approximate improvement by a factor of 6.7 compared to LPL. As shown in Section 5.3 Dozer achieves an average duty cycle of 0.128% on all nodes. Applying the values from Table 3.1 results in a mean energy consumption of 0.066 mW. Consequently, Dozer outperforms Twinkle by a factor of 6.4 and LPL by a factor of 43, respectively.

These numbers indicate a significantly reduced power consumption in Dozer compared to the other two protocols. In the following some of the reasons leading to this result are discussed. We start with LPL: Low-power listening suffers from several fundamental limitations. On the one hand, the constant periodic channel polls are not for free. In particular, radio switching times are not negligible. Furthermore, finding the optimal trade-off between polling frequency and preamble length requires a priori knowledge of the traffic demand. On the other hand, LPL is prone to the problem of overhearing. Communication on a link misleads all other nodes in the sender's transmission range to wake up and switch into receive mode. Nevertheless, LPL is a powerful approach in scenarios beyond the scope of ultra-low power data gathering. Its simplicity and robustness make it an excellent choice for a wide range of other applications.

Twinkle achieved a substantially better result but still performed measurably worse than Dozer. Based on the information extracted from [19] we

[1] In fact, we consider a variation of LPL called Pulse [42]. It optimizes the power consumption of LPL by listening for energy on the channel rather than a decodable preamble. This reduces the cost of listening substantially.

assume that the difference mostly stems from two protocol aspects. First, the mechanism used to associate a child with a parent requires periodic transmission of advertisement messages and channel overhearing in the length of a TDMA slot used to accept incoming connection requests. This procedure can be mapped to Dozer's beacon messages and their subsequent contention windows. However, in our system the contention window is only activated upon request and its length is considerably shorter than its counterpart in Twinkle. Second, the combination of coarse-grain time synchronization and CSMA-based collision avoidance leads to additional overhead. Even in steady state random backoffs before the start of each transmission prolong a node's radio uptime.

Since the publication of Dozer new systems for data gathering in wireless sensor networks have been proposed. We discuss two of these systems, Koala [36] and TSMP [41] as they claim to achieve similar performance results. Furthermore we briefly discuss IP 6LoWPAN [34] as it might represent one future path for sensor networks.

7.1.2 KOALA

Koala follows an antipodal approach to Dozer. Instead of decentralizing as many of the tasks as possible Koala uses a centralized authority to manage communication. The argumentation of the authors is that previous work [53, 54, 56] has shown that systems with a lot of complexity on the sensor nodes are failure prone. They conclude that to solve this problem the complexity should be moved to the base station.

The Koala system builds on the Flexible Control Protocol (FCP), which is used to manage communication paths. All communication within FCP is initiated at the sink. Each sensor node periodically sends a beacon message and checks if it receives an acknowledgement for this beacon. If no acknowledgement is received the node goes back to sleep. To start a communication round the base station listens for incommoding beacon messages and acknowledges them. Nodes receiving an ack stay awake and start to acknowledge the beacon messages they receive from their own neighbors as well. Applying this mechanism recursively the entire network is brought online. All nodes also forward their direct neighborhood towards the sink who is then able to generate a graph of the network topology and to compute paths to all nodes.

Using source routing the base station then sends a message to the first node it intends to download data from. All nodes on the path switch to a different communication frequency and establish a virtual channel forwarding all messages according to the route stored in the setup message. Once the node at the end of the path has finished uploading its data it returns to the

common management frequency and goes to sleep. The previously second last node on the path then starts to upload its data towards the sink. Once all nodes of a path have completed their upload the base station once again wakes up the entire network and starts downloading data from another path. That is, to collect data from all nodes, the network goes through multiple wake-up cycles which require the network to stay awake up for extended periods of time. In many practical scenarios where the network diameter is small (e.g. 3-5 hops) but in turn the MST is wide, the network will go through up to $O(n)$ wake-up cycles with n being the number of nodes in the network before all nodes have uploaded their data.

The Koala approach is promising for scenarios where data is not continuously collected at a sink but only on demand and at a very low rate (e.g. once a week). In [36] the beacon interval of a node in stand-by is set to 20 seconds what corresponds to a radio duty cycle of approximately 0.1%. In turn, the actual data aggregation is very expensive. Assuming a simple line topology of 10 hops and no collisions of any acknowledgements the time to wake up the entire network is on average 100 seconds. Nodes once woken-up may not return to sleep and thus nodes close to the sink have to stay awake for the entire 100 seconds. Even at a data collection rate of only once per three hours these 100 seconds of activity result in an additional 1% overall radio duty cycle. In order to limit the impact of these collection phases on the radio duty cycle it is therefore necessary to aim for an aggregation rate of less than once a day. On top of the network wake-up time come the non-negligible costs for the aggregation of the network topology and the consequent notification of the network specifying which nodes have to remain awake and are now part of a communication path. Unfortunately, the power analysis in [36] is incomplete and it is thus not possible to give a precise estimation of the total radio duty cycle for a given scenario. However, the Koala approach clearly favors scenarios in which data has to be collected at very large intervals. In this case the overhead for setting up the communication paths is compensated by the low stand-by power consumption of the network. For the same reason Koala is ill suited for "continuous" data aggregation as in this scenario the wake-up costs dominate the power budget and the performance of the system deteriorates. Another limitation of the system is the requirement for the network to remain stable over extended periods of time during which the routing path is computed and the actual data upload is executed. As discussed in Section 1.3 this basic requirement may not be matched in many real-world scenarios and Koala will spend more energy on recomputing data aggregation trees or fail completely in case of high interference.

7.1.3 TSMP

The Time Synchronized Mesh Protocol (TSMP) was proposed by Dust Networks [38] and represents the state of the art in industrial sensor network communication stacks. TSMP is part of the Wireless HART [62] standard defining the wireless counterpart to the widely used HART standard for commercial building automation. As Koala, TSMP is fully centralized but uses a synchronized communication model. All nodes in the network share a counter telling them how many slots of the global schedule have been executed since the startup of the network. According to this counter and the schedule defined by the base station each node knows when to send and receive messages. This slot counter is also used to determine the communication frequency, as TSMP uses channel hopping to reduce communication problems.

All nodes connected to the network broadcast advertisement messages which newly joining nodes use to detect the presence of a network. A new node sends a join request to the sender of the advertisement packet which in turn forwards the request to the central authority. The base station then provides the new node with one or multiple dedicated communication slot in the global schedule. TIt also defines along which paths the node is expected to forward its data and for which other nodes it has to serve as a relay. This information is distributed to all affected nodes that update their local routing table accordingly.

In order to sustain the global schedule all nodes in the network are synchronized using a simple clock sync mechanism. The clock value of the base station is periodically distributed through the network and MAC layer timestamps are used to minimize estimation errors. To optimize power consumption both a push and a pull mechanism are included for clock value distribution.

Furthermore, unlike the majority of academic systems TSMP offers the user the possibility to define that a network is now "complete". That is, once all expected nodes have registered with the network, advertisement messages can be turned off to save more energy. According to [41], a leave node can thereby reduce its idle radio duty cycle to 0.01% as it only receives periodic messages to maintain its clock synchronization.

In [41] the authors compare TSMP results to Dozer. To do so, they recreated the conditions of two different nodes in our office floor setup of the initial Dozer release. On the one hand they constructed the network topology according to a snapshot given in Section 3.3 for a node with 5 children and 13 descendants. This node had a radio duty cycle of 0.32% in our experiment. With TSMP the authors were able to reduce this duty cycle to 0.27%. For the measurement of the radio duty cycles the authors have thereby chosen a stable network topology and turned off TSMP's beacon messages. In other

words, TSMP was locked for a static topology and the protocol was no longer able to make any changes to the network topology in case of interference. In contrast, Dozer was executed with all features turned on in a "live" scenario in which no such stable topology existed. Consequently, multiple changes to the network topology where necessary each day to cope with external interference and a static network definition would have failed to collect the sensor readings.

The second comparison presented in [41] is for a leaf node. TSMP achieves a radio duty cycle of 0.02% as compared to 0.07% in the initial Dozer system or 0.05% in the latest Dozer release. Here the comparison is simple as in TSMP (once again with turned off advertisement messages) only one message is received to synchronize the node to the network. In Dozer, the parent beacon corresponds to this synchronization message. Additionally, in our system, the node also sends a beacon message of its own (which corresponds to an advertisement in TSMP) which doubles the power consumption. The remaining difference between the systems stems from the different maximal clock drift compensation, as in TSMP only 20 ppm of relative drift are handled whereas in our experiments up to 200 ppm were covered.

The option to turn off beaconing for leaf nodes is interesting as it offers a reduction in power consumption of approximately 50%. However, in a dynamic scenario where a leaf node may be required to become a relay at any point in time, turning off the beaconing mechanism may lead to massively increased energy costs for other nodes looking for a new parent or even a network breakdown in the worst case. Furthermore, leaf nodes already now spend only a small fraction of their total energy budget on radio uptime and are the last nodes in the network to deplete their batteries. For Dozer we have therefore decided that the energy spent on beacon messages at leaf nodes is more than compensated by the increased flexibility of the system.

7.1.4 IP is Dead, Long Live IP for Wireless Sensor Networks

J. Hui and D. Culler propose a different approach to communication in wireless sensor networks. They make the point that after several years of studying sensor networks and their properties it is time to work on the integration of WSNs into IP-based networks. They argue that with IPv6 a new standard suitable for adaption to the requirements of sensor networks is available. In [21] they propose a system adopting IP routing for wireless sensor networks. The system they propose consists of a MAC and routing layer, as well as mechanisms to modify properties of IPv6 to match the limited resources available on sensor nodes featuring an 802.15.4 compliant radio. The network

setup consists of sensor nodes and so called "border routers" interfacing the sensor network with an external IP network.

The MAC layer of the proposed system uses a highly optimized low power listening (LPL) strategy similar to WiseMAC [9]. The authors argue that a LPL strategy better matches the properties of an IP-based communication system than a synchronized model. They argue that IP-based systems usually expect a link to feature "always-on" and "low latency" properties which are easier to realize with a LPL mechanism than a scheduled communication.

The inbuilt routing system is based on the Distance-Vector Routing principle in that a node stores the costs to reach a border router through each neighbor. Costs are thereby computed in terms of necessary message transmissions including expected message losses along the path.

Another challenge for the system is header compression and dealing with the different Maximal Transfer Units (MTU) of IPv6 and 802.15.4 radios. The minimum MTU supported by IPv6 is 1280 bytes and the header size is at least 40 bytes. In contrast the payload of a 802.15.4 frame is limited to 127 bytes. With RFC 4944 [34], also known as 6LoWPAN, the authors propose a system dealing with packet fragmenting and header compression however, a detailed description of RFC 4944 goes beyond this comparison.

The authors of [21] also provide an implementation of their network stack including support for one UDP and one TCP connection. With a ROM footprint of 24,034 bytes and a RAM usage of 3,598 bytes this stack can be run on most current sensor nodes but especially for the low-end devices supporting only 4kB of RAM there is not much space left for application and sensor specific code. They provide numbers for a demonstration setup consisting of 15 nodes. 7 of the nodes were directly connected to the border router and the remaining nodes are within a 2-3 hop neighborhood. With one data sample per minute the network achieved an average radio duty cycle of 0.65%. Unlike claimed in [21] this is approximately six times as much energy as Dozer would require in the same scenario. However, the low-power listening approach brings the advantage of a lower per hop delay and thus on average a message is forwarded one hop within 0.125 seconds as compared to 15 seconds in Dozer.

With its novel approach of making WSNs IP compliant and thus easy to access from the Internet the proposed system may be the way to develop future WSN applications and to help the technology achieve its breakthrough. First practical studies such as [49] proof the conceptual feasibility of the system. A remaining open question is whether it is the best approach to move the entire complexity of IP, UDP, and TCP into the sensor network. Several translation steps such as header compression in 6LowPAN are inherently necessary in every deployment. That is, a border router will always have to provide some services to the sensor network and one might thus consider

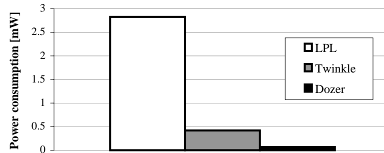

Figure 7.1: Average power consumption of LPL, Twinkle, and Dozer given a two minute sampling period.

moving more application specific code into the gateway. On the one hand, having the entire complexity directly in the sensor nodes allows the production of simple, generic border routers as they can be agnostic to the semantics of the communication between the sensor network and the Internet. On the other hand, providing the functionality of a simple application server on the gateway might drastically improve the performance of the system. For example, the answer to popular queries might be cached at the gateway and would thus not result in a lot of repeating traffic in the network. Similarly, application specific communication systems not matching the requirements of IP connections could be employed in the sensor network while still maintaining the IP-frontend to the Internet by having the gateway (proactively) storing all necessary information. Making the sensor network accessible to Internet-based services is doubtlessly the right approach and time will tell whether the fully decentralized approach or a mixed strategy will result in the best results. A more indepth comparison of the pros and cons for different methods of connecting sensor networks and IP-based infrastructure is given in [7].

7.2 More Related Work

Corresponding to the importance of the problem, there have been a plethora of research efforts addressing data gathering in the last few years. Energy efficiency of most existing work [24, 31, 56, 48, 4] stems from the application of generic energy-efficient MAC protocols [42, 58, 63]. These protocols turn off the wireless transceiver whenever possible to save power. Two types of protocols are thereby distinguished: TDMA and contention-based protocols. Protocols falling in the latter category incorporate duty cycling to achieve low power operation. [63] and [30] coordinate the nodes' sleep schedules such that neighboring nodes are awake at the same time. In the active phases CS-MA/CA is used to control channel access. To achieve high energy efficiency the active periods must be very small compared to the time nodes are in sleep

mode. Since the whole network wakes up at roughly the same time nodes suffer from high channel contention which reduces network throughput. T-MAC [58] is an improvement of S-MAC [63] handling varying traffic load with adaptive duty cycling. The protocol does however not overcome the inherent limitations of this approach. *Low-power listening* is another strategy to condition contention based MAC protocols to low-power requirements. To avoid idle listening nodes turn off the radio most of time, only periodically probing the channel for the presence of activity. Once network activity is detected the node switches on its radio to listen for the incoming packet. To ensure the receiver is listening a sender has to prefix its packet with a long preamble acting as an in-band busy-tone. A key advantage of asynchronous low-power listening protocols [42, 9] is that the sender and receiver can be completely decoupled in their own duty cycles. However, these protocols suffer from the overhearing problem, since the long preamble also wakes up nodes who are not the intended receiver of a packet. To overcome this drawback [64] proposes to synchronize the channel polling times of all neighboring nodes, thus preventing the protocol from sending long preambles. This move incurs contention during the scheduled channel probing which is resolved by using CSMA. A drawback of this protocol is that all nodes require to be tightly synchronized to meet energy efficiency which creates additional costs.

In contrast to the aforementioned protocols, TDMA-based solutions establish a schedule where each node is assigned one or possibly multiple time-slots. In each slot nodes are then able to communicate without provoking packet collisions or suffering from overhearing. Pure TDMA protocols are however hardly feasible in reality since they require global time synchronization and are susceptible to topological changes of the network. Hence, most proposed protocols use a combination of pure TDMA and the above mentioned contention-based approach.

In [45] a two phase protocol is proposed. In the first phase a node collects information about its two-hop neighborhood and participates in a distributed slot allocation procedure. In addition, a protocol for network-wide time synchronization is executed during this phase. Once the TDMA schedule is computed in the first phase the protocol switches over to the second phase where the schedule is executed. DMAC [28] proposes an adaptation of S-MAC optimized for data gathering. The protocol assumes that a routing tree towards the data sink exists. The active periods of the nodes are staggered according to their level in the tree. CSMA is used to arbitrate between children in order to prevent collisions. DMAC achieves low data delivery latency at the sink. However, there is a substantial overhead in case of network instabilities and due to the local synchronization at the nodes. FPS [20] and its descendant Twinkle [19] are closest related to the protocol described in this paper. The coarse grained scheduling of FPS represents

a distributed TDMA approach where each node schedules its own children. Although this schedule ensures that parents and their children are contention free, collisions may still occur due to other nodes in the network or poor time synchronization. This contention is handled using CSMA. The protocol does not incorporate a tree construction and is thus dependent on other protocols establishing such a network topology. In contrast to our solution FPS—and thus also Twinkle—requires global time synchronization.

Chapter 8

Concluding Remarks and Outlook

With Dozer we have proven the viability of time scheduled communication in multi-hop wireless networks at very low radio duty cycles. For the application scenario of long-term environmental monitoring our protocols achieve duty cycles in the order of 0.1% while maintaining full self-organization and self-repair abilities in the network. Several productive deployments of Dozer are in operation by Shockfish and the PermaSense project successfully employs our communication system for their deployments in the Swiss Alps.

Dozer is designed and optimized for low data rate data aggregation applications where delay is of limited importance. For this class of applications we do not see much more optimization potential. Tweaking parameters such as the parents update mechanism and the minimal guard times according to the environmental conditions of a specific deployment may lead to a slight decrease in the radio duty cycle but the saved energy will not be significant.

However, there are related scenarios for which Dozer is currently unsuited but where modifications to the protocol might be interesting. For example, by breaking up the current fix data rates and slot lengths in the TDMA schedules one could construct a protocol offering variable data rates on different links. Connections between leaf nodes and their parents could be slowed down so that uploads only occur at the same rate as the sensor is sampled. Similarly, links closer to the sink might benefit from faster schedules to forward their buffered messages. Besides an increased message throughput, such a mechanism might also lower the end-to-end delay messages incur in the network.

Other possible adaptations include leaving the many-to-one communica-

tion scenarios and to allow any-to-any message exchanges. Such a change would significantly change the operating parameters as meta information about more links would have to be stored on the nodes. Memory becomes a bottleneck and additional mechanisms deciding on the priority of communication links have to be developed.

Another open issue is the question of the optimal tree topology. In general multiple data aggregation overlays can be built on a physical network. At the moment Dozer tries to find the most stable, shallow tree as the hop distance to the sink is the primary attribute according to which a node chooses its parent. Intuitively this is a reasonable choice in many scenarios but under certain conditions it may lead to an undesirable, unbalanced traffic distribution in the network. Incorporating other factors in the parent selection mechanism, such as energy left in the batteries or the current message rate, these cases might be detected.

Beyond the possible modifications to the protocol we also learned from the PermaSense project that an implementation as complex as Dozer is difficult to use for external developers. Without in-depth knowledge of the system it is hard to incorporate our code in a full-fledged application. A set of additional tools could help alleviate this problem. Such tools could include configuration wizards helping to set the numerous parameters through which Dozer can be fine-tuned but also include application examples showing how to embed the communication stack and how to use the command dissemination mechanism.

Part II

Developement Support for Wireless Networks

Chapter 9

Introduction

Parallel to the development of the Dozer project we were interested in how to improve the development cycle of distributed applications in ad hoc and wireless sensor networks. At first, the focus was more on ad hoc networks consisting of larger devices featuring wireless LAN interfaces but over time—and with our growing experience in this area—it moved towards wireless sensor networks. Although these two classes of networks both fall in the category of wireless networks, we learned that their development requirements are often different. Computation power, available memory, employed programming language, and underlying operating system are some aspects leading to these differences. Another important difference influencing the development cycle of applications is the available user interface. While PDAs, cell phones, or notebooks offer a wide range of possibilities to give feedback and status reports to a user this is not the case for wireless sensor nodes. Three LEDs are oftentimes the only directly accessible information a human can get from a sensor node. If this is insufficient, serial communication or over-the-air transmission of debug information becomes necessary.

At the outset of this dissertation the development process for wireless sensor network applications bore a close resemblance to writing programs for personal computers in the 1980s: A simple text editor for programming, shell-based compilation and flashing of the applications, and communication logs in the form of hex-dumps were the tools we had to work with. This total lack of development support was one of the reasons for the long development cycle of the Dozer code. We not only had to write the communication stack but also a set of monitoring and analyzer tools. From discussions with other researchers in this area we found that this is a common problem and even the most basic support tools were re-developed over and over again for each

new project. We therefore decided to investigate how to build generic development support tools that help speeding up the development process of wireless sensor network applications.

In the second part of this thesis we present support tools that we have developed over the last years. We discuss the advantages but also limitations of our applications and try to evaluate their impact on the community they were designed for.

Chapter 10

Simulation of Ad Hoc Networks

The question of development support for wireless ad hoc networks arose for the first time during a course for graduate students called "Mobile Computing". In this course the students were expected to develop an instant messenger application in Java. To make things a bit more challenging the messenger system was completely server free, communication was based on IP multicasts, and a multi-hop routing scheme for communication with other users out of direct transmission range was to be developed. We provided the students with a set of routing and application level protocols which they had to implement during the semester. At the end of the course we organized a get-together during which the students had the chance to test if their applications where compatible to each other and to see if they could exchange messages.

Previous years had shown that the students were generally quite excited with this semester-long exercise. However, as soon as the multi-hop part of the exercise started they ran into difficulties. One of the most prominent problems turned out to be the challenge of testing and debugging the application. Single-hop communication was easy to test by working in small teams but as soon as message relaying was required things became more complicated. Wireless LAN adapters have a reach of several meters and thus even a minimal two-hop communication experiment required at least three people to spread over a whole office floor; a rather inconvenient way of testing an application. Thus, many students either gave up on the exercise or handed in untested solutions which often crashed under live conditions.

To improve this situation we searched for development tools that might

be of help. Especially, simulation and emulation tools were of interest to us. At first we evaluated well known simulation systems such as NS-2 [10] and GloMoSim [65]. These systems aim for perfect simulation of all network layers and are consequently very complex. Due to their complexity these tools bring the risk that improper configuration of the simulation environment may lead to erroneous or misleading results. Furthermore, if only one or few layers of the network stack are to be evaluated, employing a full network simulation is like breaking a fly on the wheel. For educational purposes—such as in our case—the complexity of these network simulators turns into a problem as the effort and time necessary to understand the simulation environment often stands in no reasonable relation to the yielded benefits.

Another problem with simulation environments is that they usually require code to be written in a product specific language. In the end an implementation of our instant messenger for the simulator would only run within the simulation tool and not directly on a notebook. This was contradictory to our intent of increasing the students' awareness of real-world problems faced when developing algorithms and applications for wireless networks. In the end we found that there was no simple to use solution available to test an application using multicast communication in Java. This was surprising as with the constantly increasing number of devices supporting WLAN and Java—especially cell phones—this specific method of communication seems very promising for applications involving interaction between groups of people in one room. Particularly, in the domain of social networking IP multicasts could be used to connect crowds of people in the same area.

10.1 Simple Ad Hoc Network Simulator

Against this background we introduce SANS, a Simple Ad Hoc Network Simulator with the goal of providing an intuitive and easy to use emulation tool. Ideally, five minutes should suffice to become familiar with SANS as a development, testing, and debugging tool.

One of the main reasons for SANS' simplicity lies in its focus on the simulation of network and transport layer protocols. Both of these layers deserve special attention in the context of ad hoc networks as wireless links are much more dynamic than traditional wired connections. For example, packet loss rates are much higher and thus routing and transport layer protocols have to be designed to be able to cope with lost messages. To help developing such protocols SANS provides an abstraction of the physical and data link layers. In particular, a program running in the simulator can use the Java UDP multicast interface to send and receive data packets from its direct neighbors in the simulated ad hoc network. An advantage of this approach is that

applications running in SANS can directly be executed on any Java-enabled system as long as the platform supports the Java UDP multicast interface.

Since SANS is not only intended for prototyping but also for educational purposes, Java has been chosen to implement the simulator itself and also as the programming language used to write the simulated programs. As a direct consequence the simulator is platform-independent and can be employed on all operating systems supporting the Java environment. Furthermore, it is possible to run the simulation from within a Java IDE such as Eclipse [55] and to make full use of the debug facilities of the IDE. It is therefore possible to use features such as breakpoints or variable introspection in the application running on a simulated node.

10.1.1 Design Goals

Our goal was to design a system which does not require more than a couple of minutes to get used to but still offers all necessary options for testing and presenting algorithms running on wireless networks. The key features we defined for our system can be summarized as follows:

- Ease of use,
- graphical representation of the network,
- real-time network simulation,
- code developed on the simulator should run on hardware without adaptation,
- individual transmission ranges for each node,
- adjustable link properties such as delay and packet loss, and
- support of a standard programming language.

In the following sections we will give an overview of how we realized these design goals.

10.1.2 Overview

SANS simulates an entire network of independent nodes on one computer. The simulated nodes are thereby completely isolated from each other and do not have to execute the same application. As shown in Figure 10.1 SANS renders a schematic representation of the network including nodes and possible links in its main window. Network nodes running arbitrary Java client applications are displayed as circles with a node identifier in the center. Links

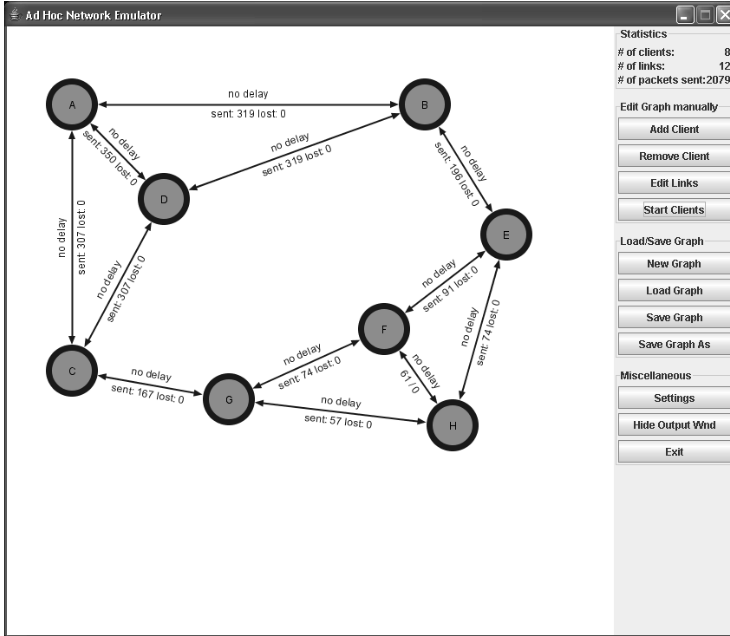

Figure 10.1: SANS main window showing a network of eight partially connected nodes.

indicating that two nodes are within (mutual) communication distance are symbolized by arrows. Whenever a message is sent over such a virtual link the head of the corresponding arrow flashes for a moment and the packet count of the link is increased. In many cases this visual feedback suffices to observe the behavior of an algorithm without the need of additional log files and similar post processing techniques.

Setting up a network is a matter of a few drag-and-drop operations and a few small dialogs for entering parameters. A right-click on the main window creates a new node at this position and a dialog pops-up where the user is asked to specify the client application which is to be run on this node. Similarly, a virtual link between two nodes is established by dragging a line from one node to the other. The dialog shown in Figure 10.2 opens and the user is asked to specify the link properties. SANS supports one-way and bidirectional links. Thus, it is possible to simulate networks consisting of heterogeneous hardware with different transmission ranges. Also delay

and packet loss can be set individually for each link. We have explicitly decided against using a model-based approach to define which nodes are within mutual communication range. The reason for this decision is that practically all models, from the simple Unit Disk Graph (UDG) to complex Signal to Interference and Noise Ratio (SINR) systems, expect transmission ranges to be symmetric in all directions. Especially, for indoor setups where walls and other obstacles are blocking direct line of sight between nodes it is easy to come up with a scenario where a circular transmission range is unreasonable. However, also on a plain field communication ranges are not identical in all directions [22]. Differences may be caused by the design of the employed antennas or external interference. Thus, all simulations based on such connection models make assumptions which do not map well to real-world deployments. While the simulation of networks with hundreds or even thousands of nodes can only be realized by having an algorithm place and connect the nodes, small scale experiments such as the ones possible with SANS can easily be configured by hand. We therefore decided to prioritize the possibility of building tricky topologies (e.g. with zones of high interference) over a more convenient model-based topology generation.

Another feature worth mentioning is the possibility of topology changes at runtime. Links and nodes can be added or removed while the simulation process is running, without the need of a restart. Hence, SANS can be used to test the reaction of an algorithm to spontaneous network topology changes which are a common problem in wireless networks. Of course SANS also provides convenient access to console output produced by the applications running on the simulated nodes. The output of each individual node is shown the GUI and also the generation of log files is supported. In combination with the inbuilt message history it is possible to reconstruct the exact course of a simulation run which may be necessary when debugging an application.

10.1.3 Simulation of Physical and Data Link Layers

SANS is designed to help evaluating transport layer and routing algorithms. Hence the system has to provide a simulation of the lower layers of the communication stack. Following our demand for simplicity we decided to build an overlay network on top of the UDP multicast system. Multicasts feature similar properties as radio communication as not only the intended recipient receives a message but all members of a multicast group. In SANS all nodes join the same multicast group and thus, every message sent through the system is received by all nodes in the network. This behavior maps to a network where all nodes are within mutual communication range or a fully connected topology if we think in terms of network graphs. This model is therefore unfit for testing multi-hop communication algorithms. To overcome this limita-

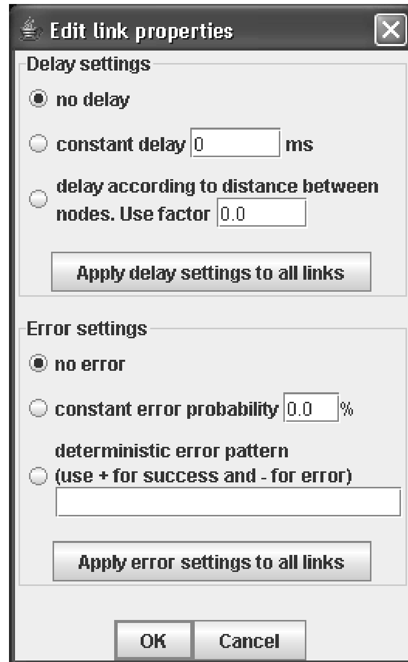

Figure 10.2: Configuration dialog for link propertiy definitions.

tion, SANS uses a firewall-like mechanism to limit message delivery to nodes directly connected to the sender in the schematic network representation.

The main advantage of building an overlay network on top of a UDP multicast group is the existence of a handy interface in Java dealing with multicast communication `java.net.MulticastSocket`. SANS intercepts calls to this interface and handles them internally. This process is completely transparent to the user and the application executed on a simulated node. For developers this brings the advantage of a well-known and documented interface to send and receive messages. Furthermore, the multicast interface is supported by all Java virtual machines down to JavaME running on handheld devices. Due to this widespread support for the communication interface and the transparent interaction with SANS applications developed within the simulation environment can also be executed on real hardware without any adaptation. Method calls which are intercepted when run in the simulation environment are then handled by a physical network device.

Figure 10.3 depicts the process performed when a message is sent in SANS. The communication is initiated by a client application calling the `send` method of the Java MulticastSocket to broadcast a packet. This method call is intercepted by SANS and instead of delivering the data to a networking device it is processed by the simulation system. First, SANS determines which nodes are within transmission range of the sender (which means that there is a link in the schematic network representation between the sender and these nodes). In a second step the individual link properties of each of these connections are analyzed. According to the user-defined error model, the system decides if the message reaches the receiving node or if it is dropped due to simulated packet loss. Finally, if the message is not dropped, SANS calculates the time of arrival of the packet according to the link delay and puts the data in the delivery wait queue of the receiving node. Eventually, the receiving nodes call the `receive()` method of the MulticastSocket to poll for newly available data. Again, this call is intercepted and SANS returns the next valid packet from the corresponding wait queue.

10.1.4 Internal Network Simulation

In order to be able to run arbitrary applications in SANS the transparent interception of radio calls and strict separation of the namespaces of different simulated nodes is of fundamental importance. We exploit Java's class loading mechanism to achieve both goals. Each simulated node is started in a new thread and with a new custom class loader. Although SANS and all simulated nodes run within one instance of a Java virtual machine these different class loaders provide individual namespaces to their appointed simulated node. It is therefore guaranteed that there are no hidden communication channels be-

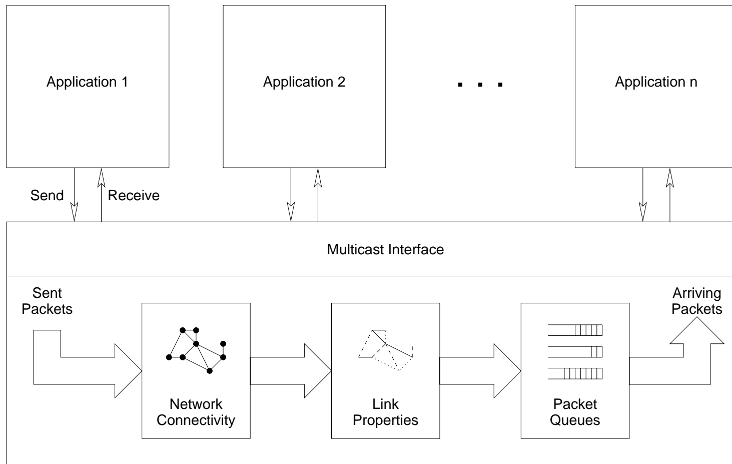

Figure 10.3: Architecture of the network simulation layer.

tween the simulated nodes (e.g. by means of static variables). Furthermore, we have full control over these class loaders and are able to hook into the Java network code. To do so the `setDatagramSocketFakapictoryImpl` method of the `DatagramSocket` class is used. The `DatagramSocket` class represents the base socket for all UDP communication in Java. By setting a different factory we force Java to return our own Sockets instead of the default implementation. Since this new factory is set before the first class of the simulated node is initialized, we are certain to catch all calls from the simulated application. As a side effect of this solution SANS not only supports communication by multicasts but all UDP communication is simulated properly.

10.1.5 Related Work

Various very powerful network simulators are available which offer full network simulation. Among the most prominent ones are NetSim [18], GloMoSim [65], and ns-2 [10]. From a feature perspective these simulators play in a totally different league than SANS and support far more complex simulation setups. Their disadvantage is their usage complexity and we built SANS specifically for the niche of comparatively simple simulations where ease of use is a major requirement.

Another interesting approach to simulation and validation of network algorithms is presented in Sinalgo [12]. Instead of simulating the different

layers of the network stack, Sinalgo focuses on the verification of the network algorithm by abstracting from the underlying layers. Using a message passing model to simulate communication it can be used to quickly evaluate the performance of an algorithm under different conditions. Sinalgo therefore aims at a different aspect of algorithm design as it does not help testing an implementation of the protocol as it is the case in SANS, but helps verifying the correctness of the underlying algorithm.

Another class of network testing tools are network emulators like the APE testbed [29], JEmu [13], or EMPOWER [39]. Unlike in simulation tools these emulators use networks consisting of several computers to evaluate a realistic implementation of an algorithm. In JEmu and APE, the topology of the underlying network is a simple star with the simulated clients running on the peripheral nodes whereas the management component of the emulator is run on the central instance. The management component controls the communication flow between the client machines making it possible to emulate various virtual network topologies. EMPOWER follows a more sophisticated decentralized approach to emulate the network. The benefit of the decentralization is a better scalability at the cost of a more complex network setup. A drawback of these network emulators is that multiple executions of the same experiment may result in different results, as the communication between the physical test machines is non-deterministic.

SANS simulates test networks on one computer but supports the standard multicast interface (`java.net.MulticastSocket`) for communication between clients. Consequently, implementations of algorithms developed with SANS can easily be ported to run on one of the more sophisticated network emulation systems or even on the real-world target platforms for closer evaluation.

10.1.6 Concluding Remarks

SANS is a simplistic tool and its feature set cannot be compared to any of the well known simulation systems. It was never our goal to create a solution for high performance and complex simulations. For these scenarios the simulation frameworks referenced in the Related Work Section 10.1.5 are far superior. However, when we look at the problem of how to raise the awareness of students towards problems of distributed systems SANS is an interesting option. The training time for new users and the amount of necessary help to get started with SANS is minimal. Developers with basic Java knowledge will be able to run their first simulations within a couple of minutes.

In our case we saw a significant increase in handed in solutions for our multi-hop instant messenger exercise once we started using SANS for the

lecture. But not only had the quantity of the handed in work improved but also the quality. In previous years many of the messenger applications failed to communicate with implementations of other students. In contrast, with SANS as developments tool the solutions of our students were more thoroughly tested. Thus, the majority of them were able to hand in a working solution respecting our protocol specific specifications and thus to exchange messages with other implementations.

From a didactical point of view it was an important step to give our students a better development tool helping them overcome the difficulties of testing a distributed application. Especially for the more interested students it used to be frustrating to work on the exercises for several weeks and in the end not to achieve the success of a working solution. With SANS we were able to give them the tool they needed to test and debug their solutions. The feedback from the students at the end of the semester confirmed that they now really enjoyed the messenger exercise and some of the handed-in solution far exceeded the functionality we had asked for.

Chapter 11

Monitoring Sensor Networks

Monitoring wireless sensor networks is a difficult task as usually there is no direct access to all deployed nodes. During the development of the Dozer system we quickly started to miss several tools which could have saved us a lot of trouble while debugging and benchmarking the protocol. For example there was neither a way to query the state of system variables in a specific node nor a simple mechanism to log and store certain values. As these are common requirements for the development of wireless sensor network applications, we have decided to design a generic framework providing support for the following tasks:

Remote Procedure Calls (RPC) to execute functions on different nodes in the network.

Logging of arbitrary data directly in the flash of the sensor node and the possibility to remotely read out this storage.

Topology monitoring providing information about which nodes are in mutual, physical connectivity range.

Topology control allowing the blacklisting of individual physical connections and thus to form a different logic network.

We built this tool including several student theses on the Eclipse [55] framework as Eclipse was well documented and open to extensions. Unlike in Dozer we decided not to build all involved components from scratch but relied on existing libraries such as the "Drip and Drain" module for data dissemination and collection in the network.

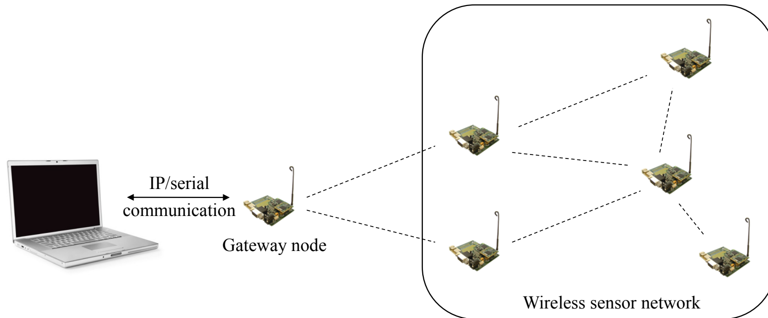

Figure 11.1: Setup of the control and monitoring application. An Eclipse-based application is executed on a work station providing the user interface. A gateway sensor node is attached to this workstation. The sensor network side consists of the deployed sensor nodes running TinyOs code.

11.1 Overview

The controlling framework consists of two logical components as depicted in Figure 11.1: The "client side" is a personal computer with an attached gateway sensor node. The gateway node thereby runs a simple application forwarding all data messages received from the work station to the sensor network. Analogously, messages received from the sensor network are forwarded to the work station. The "sensor network side" represents the deployed sensor nodes running the application to monitor.

11.1.1 Workstation Application

The user interface is fully integrated in the Eclipse software development framework (also see Figure 11.2) and offers five different views:

- The navigator view is used to manage all involved files.
- The editor view shows the connectivity map of all deployed nodes. Arrows indicate that two nodes are within mutual communication range.
- All available nodes are listed in the outline view. Opening the entry for one node lists all their installed modules, loggers, available variables and RPC commands.

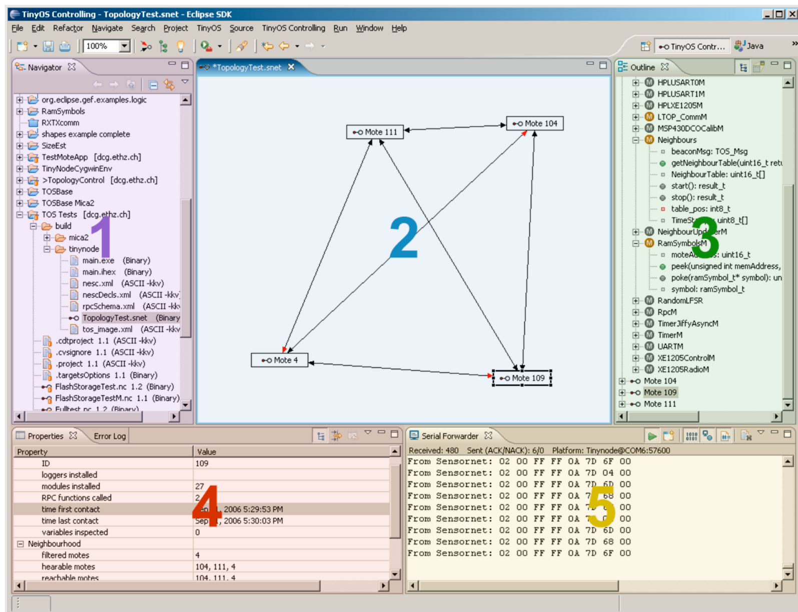

Figure 11.2: User interface on the workstation.

- The properties view provides detailed information about a selected item. Depending on the context information about the node, module, logger, RPC command, or current value within a variable are shown.

- The serial forwarder view is used to setup the connection between the work station and the gateway sensor node. It also provides a raw view of the incoming and outgoing traffic on the gateway node.

11.1.2 Sensor Network

The sensor network code consists of multiple TinyOS modules providing the necessary functionality to monitor and control the operation of a deployed sensor node. To save resources, a developer may customize the system by only including the components which are required for the current task. If, for example, no loggers are required, the entire logging facilities can be excluded.

11.2 Remote Procedure Calls

Remote procedure calls are essential for a monitoring and control application to execute support code on the nodes. For this project we decided to rely on the Marionette Pytos [61] library developed at UC Berkeley which represented the state of the art at that time. Pytos offers RPC functionality for a sensor network. At compilation time a custom XML file is generated containing details about the built application, including information about the used modules and variables and where to find them in the node's memory. Furthermore, Pytos includes TinyOS modules allowing the execution of specific functions on the node and to read and write to global variables. Furthermore, Pytos uses the Drip and Drain library as a multi-hop communication stack. Drip is an epidemic protocol used to disseminate messages through the entire network. Pytos uses Drip to broadcast RPC command to the nodes. Drain complements Drip in that it offers data aggregation functionality from the network to a base station. Drain relies on a data gathering tree rooted at the sink and all nodes store some information about their position in this tree.

As the name indicates Pytos relies on Python scripts which we have included in our environment. That is, the Pytos functionality is available to the user directly from within Eclipse and no additional command line tools or installations are required. To make a function available to RPC access the only requirement is to tag the declaration of the function with "rpc()". This tag is used at compilation time to include the function in the XML file describing the application. Our Eclipse plug-in parses these XML files and lists all available RPC calls and global variables in the outline view of a node. To execute a RPC or to read/set a global variable the user can simply click on the corresponding function in this view and gets a direct feedback in the properties view. One of the costs incurred by the Pytos system is its non-negligible footprint in terms of memory usage and especially program size. We have therefore decided to build our own extensions around the available Pytos functions. For example, we also use the Drip and Drain modules for all communication in the network as this communication stack has to be included anyway to enable RPC.

11.3 Logging

With the RPC functionality in place we started to think about how to monitor the state of a node over an extended period of time. At high resolution, real-time data aggregation becomes unfeasible as the low data rates supported by the radios employed on sensor nodes do not suffice to efficiently collect the generated information. We therefore included a logging functionality in our tool that uses the inbuilt flash memory.

11.3.1 Design Goals

We defined the following design criteria as essential for our logging component:

Persistency: Log entries have to be stored persistently on the nodes. That is, in case of a node crashing or rebooting as few log messages as possible must be lost. Furthermore, existing log entries must not be overwritten on node reboot but new log messages are to be appended at the first free position in memory.

Memory Management: The memory footprint of the logger module must be kept as small as possible. Additionally, to grant other components of the application running on the sensor node access to the flash memory, the available storage a partitioning of the available resources has to be supported.

Limited Flash Access: The inbuilt flash storage of a sensor node uses EEPROM technology. As access to this memory is expensive in terms of energy and since an EEPROM chip only supports a limited number of erase-write cycles, access to this memory must be minimized.

Multiple Loggers: To support the simultaneous observation of different parts of the sensor node application multiple, independent loggers are to be supported.

Remote Access: To prevent the need of collecting all nodes to read-out their logged data a remote read out of the stored information has to be supported.

11.3.2 Logger

As with the remote procedure call functionality we tried to reuse existing system libraries. However, due to the more detailed requirements for flash management no system matching our demands was found. We therefore built a new flash management component only relying on low-level TinyOS flash access routines. To deal with the tradeoff between minimizing the number of write accesses to flash memory and the persistency of the logged data we let the developer decide how many entries to cache in RAM before flushing this buffer to EEPROM. A large buffer reduces energy consumption and prolongs node life-time but there are also a larger number of logged entries lost if a node crashes and reboots.

EEPROM memory has the inherent limitation that it can only be written in pages. That is, for the commonly used AT45DB041 chip every write

operation writes 264 bytes to flash. Selecting a buffer smaller than an EEPROM page size should therefore be avoided if possible to prevent flash access overhead. However, for applications where the loss of log entries is unacceptable, a smaller buffer size and thus more frequent flush to EEPROM may be preferable. A buffer size larger than the EEPROM page size is unadvisable since as soon as the buffer exceeds the bounds of a flash page it has to be written anyway. Therefore, no write accesses are saved but more buffered log entries get lost if the node reboots.

We use a simple file system consisting of meta information at a well known address in flash memory to organize the persistent memory. For each logger there is an entry defining its starting position in memory, the length of one log entry, and the amount of memory pages to reserve for this logger. With this information it is possible to grant each logger its own share of the flash memory and to prevent collisions. An individual logger is implemented as a circular queue that will overwrite the oldest entry if it runs out of memory.

Besides the meta information about the logger itself each page of log entries is trailed by four bytes of meta data including a sequence number and the number of log entries in this page. It is therefore possible to find the end of the current log also after a reboot by traversing the pages of the flash store until an inconsistency with the page numbers is found.

11.3.3 Remote Log Reader

Logged data can be retrieved on a push or pull basis. In push mode, a logger is configured to send its log to the workstation as soon as a certain number of log entries were generated. Especially for long term surveillance this mode is preferable as no data loss due to overflows within the queue of the logger may occur. Pull mode is best suited to get a live update of the current state as it is often required during debug sessions. The developer can access all installed loggers by opening the entry of a node in the outline view of Eclipse and then requesting the download of the stored values of a specific logger.

The actual data transfer from the nodes to the client is handled by the Drain module, which is also used for RPC. Drain does not handle message losses on the wireless channel and therefore the log received at the client may be incomplete. Due to the employed sequence and transaction IDs it is however possible to identify the missing packets and to individually request them for retransmission.

On the client side several tools for log handling are integrated in Eclipse. A live view of the received data in its raw state as a hex dump is available in the console and generation of a log file is also supported. Furthermore, we have integrated a database interface allowing the insertion of log entries into an SQL database. As the schema of the database used to store logged data is

application dependent the developer is asked to provide corresponding insert statements for received log data. The arriving raw data is parsed according to the logger meta information and the insert statement is called for each individual log entry.

11.4 Topology Control

The last major component of the control and monitoring application is its topology control module. Multi hop setups are often difficult to construct, especially if specific topologies are required. A commonly employed trick is to encode topology information in the testing application. That is, each node gets a white or black list defining which other nodes are acceptable communication partners and which other nodes are to be ignored. Using this approach multi hop networks can be constructed while keeping all nodes in a close area. The results of such a testbed are not 100% realistic as the blacklisting of certain communication channels is purely virtual and thus undesired physical effects such as packet collisions happen at a higher rate than on a real world deployment with a less dense node distribution. On the other hand, as all nodes are kept in close vicinity to each other they also witness similar environmental conditions and are thus for example not forced to compensate for different ambient temperatures. Nonetheless, such a virtual testbed is a great tool for initial testing of an application and more reliable than simulations.

11.4.1 Physical Neighborhood

As a basic service of the control application all nodes periodically send a beacon message containing information about them. All nodes in reception range add the sender of a received beacon message to their list of physical neighbors. This list can be accessed from the Eclipse application (see Figure 11.3) to draw a graph of the physical network topology. Furthermore, nodes automatically notify the client if their neighbor information changes, that is if they hear a beacon from a new node or if they fail to receive multiple consecutive beacons from a previously found neighbor.

11.4.2 Virtual Overlay

To enable the topology control component on the sensor nodes, TinyOS 1.x' default communication interface `GenericComm` has to be replaced with a custom module providing the same interface as `GenericComm` but which also incorporates a firewall like component we use to block specific messages.

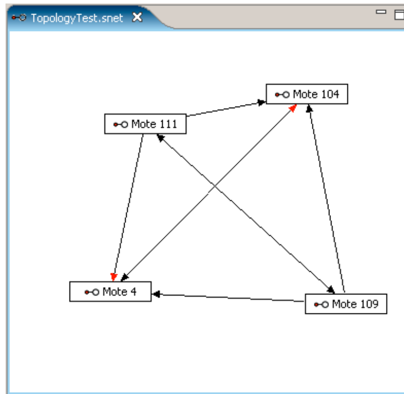

Figure 11.3: Graph view of the physical communication ranges and blacklisted connections $node111->node4$ and $node4->node104$.

This replacement can be hardcoded by directly linking the new module or more elegantly by using a precompiler statement. The second approach has the advantage that the full functionality of the blacklisting mechanism can be turned on/off at compile time by enabling or disabling a single command in the make file.

The topology control module on the nodes maintains a blacklist of neighbors from which the application is not allowed to receive messages. The originator of every physically received message is compared to the blacklist and in case of a match the message is destroyed without notifying the application running on the node. To modify the blacklist a set of simple RPC functions is available. A developer can modify the list of a node directly within Eclipse by using point and click operations on the graph rendering physical topology of the network.

11.5 Concluding Remarks and Outlook

The controlling and monitoring framework was our first major project in the area of development support for TinyOS applications. At the start of the project, several promising command line tools with more or less complicated installation and usage requirements were available. We decided to reuse as many of these existing libraries as possible but to hide their complexity from the developers by giving them a user-friendly interface to all available functionality.

The project successfully imported the the Pytos library, which was the most powerful remote control library at that time. Building on Pytos allowed us the rapid development of multiple control and monitoring features. The combination of topology control, logging, and remote introspection of the node state sums up to a powerful tool enabling the observation of a program at a level of detail usually unavailable to sensor network developers.

One of the most critical decision we had to make was whether to include a custom communicaion stack or to tap into the communication system implemented by the application we are monitoring. We decided to include a custom communication system for two reasons: First, not all applications necessarily contain data aggregation functions and therefore it was not certain that a suitable communication stack would be available in all applications. Second, the integration of Drip and Drain in Pytos was hardcoded at several places. It would have been possible to strip the communication stack from Pytos and replace it with another system but due to the tight integration of the two systems we feared negative side effects. We therefore decided to also rely on Drip and Drain for the additional functionality we were implementing.

In practice the system works well for small to medium-sized networks with up to a low two-digits number of nodes. If larger networks are monitored the communication overhead of Drain becomes a nuisance as the tree maintenance results in a lot of flooding and echo messages. Furthermore, the program size for the full-fledged application including RPC, logger, and topology control fills up the most part of the sensor node's ROM.

As already the authors of Marionette Pytos have stated, the communication overhead of Drip and Drain is often acceptable during the development and debugging phase of an application. However, for the final deployment the system is too energy intensive to be used. As a future work we therefore see the integration of a more energy aware communication stack. The challenge will thereby be to reduce power consumption while maintain a certain level of responsiveness in the system.

Chapter 12

YETI: An Eclipse Plugin for TinyOS Development

12.1 Introduction

Since different fields of applications for sensor networks oftentimes require specific hardware, a lot of work has been spent on building highly efficient nodes. Various node families have been developed with different design goals and target applications in mind. A perfect node, ideal for all tasks cannot be designed. However, today's platforms offer sophisticated solutions for most requirements, be it a large set of preinstalled sensors [47], simple extensibility [5, 17], a long transmission range [17], multiple on-board radio devices [2], or an integrated USB interface [35]. Also TinyOS, one of the most prominent operating system for sensor networks, is under constant development. An active community of developers improves the system's performance and extends its functionality.

Despite the large user base and the advances in hardware and software, development tools for TinyOS are still rare and often severely limited in terms of functionality. Hence, TinyOS developers are forced to use generic text editors for writing their applications. A command line shell is required to compile code and flash nodes with the resulting binaries. The lack of convenience functions such as real time spell checking, code completion, or even a correct syntax highlighting makes the development of TinyOS applications an unnecessary cumbersome task.

With YETI[1], we provide an Eclipse plug-in offering support for TinyOS

---

[1] **Y**ETI is an **E**clipse based **T**inyOS **I**DE

development from within the Eclipse framework [55]. YETI provides all important features known from development environments for other programming languages and is designed to be of use for both, inexperienced and professional sensor network developers.

12.2 Development Requirements

From our own experience and numerous discussions with other TinyOS and sensor network developers we found that the requirements of TinyOS newcomers and experienced developers vary. Newcomers who have little experience in writing sensor network applications need help on fundamental aspects of TinyOS development. For one, getting used to the design philosophy of the operating system is not easy. Especially its completely modular application design and the unique way of combining modules by means of a so called *wiring* require some time to get used to. The well written tutorial on the TinyOS homepage helps to overcome this steep learning curve since all important features of TinyOS and its programming language *nesC* [15] are discussed. Still, it is not unusual for the first contact with TinyOS to be discouraging. The installation of the system and the necessary toolchains often leads to problems. If the provided installer fails, repairing a new TinyOS installation requires a sound knowledge of the system which inexperienced developers have not yet achieved.

Another problem for new TinyOS developers is the vast amount of files included in the sources of the system. TinyOS features numerous modules solving many common tasks. Alas, it is often difficult to find the correct files providing the required functionality and thus newcomers show a tendency to ignore them. Experienced developers are more aware of these sources. Yet, in spite of their in depth knowledge of the system they spend a significant amount of their development time browsing through various TinyOS directories looking for adequate implementations of the functions they need.

For more ambitious projects also aspects such as rapid prototyping, cross platform development, and support for backup and version control systems are of greater importance. Furthermore, the possibility of having several parallel installations of the TinyOS source tree is critical for many developers. On the one hand a snapshot installation provides a stable development environment while on the other hand a Concurrent Version System (CVS) checkout of the operating system allows using bleeding edge technology which has not yet made it into the stable release.

12.3 Features

The goal of YETI is to provide an efficient development tool for experienced users and a convenient, easy to use environment for newcomers. Consequently, all requirements mentioned in Section 12.2 have to be considered. Also aspects such as Look-and-Feel are of importance if a large number of users is to work with the tool. We therefore decided to build YETI on top of the widely used Eclipse framework [55]. Eclipse provides a powerful plug-in mechanism allowing nearly unlimited extensions and enhancements of its inbuilt functionality. Due to this ease of extensibility Eclipse has become first choice for many developers and plug-ins supporting various programming languages such as C(++), Fortran, or Cobol have been written. Furthermore, Eclipse is designed to allow easy incorporation of existing features in a new plug-in. Consequently, YETI benefits from various existing Eclipse components such as the basic editor, the persistency system, or the CVS client. Also updating the plug-in is possible using Eclipse's update mechanism.

YETI consists of two Eclipse plug-ins: The *System* plug-in containing the functionality of the development environment and the *TinyOS Environment Wrapper* plug-in providing access to a TinyOS installation. In the subsequent sections we will discuss these two plug-ins in more detail.

12.3.1 System Plug-in

The System plug-in provides the actual programming environment and tools for TinyOS development in Eclipse. As can be seen in Figure 12.1, once YETI is installed a new custom *TinyOS perspective* becomes available. This perspective is optimized for the task of writing sensor network programs and hosts helpful features for all stages of development.

Project Creation

New projects are created using the *TinyOS project wizard*. In a short dialog the user can name the new project, choose one of the available TinyOS installations (also see Section 12.3.2), and define a default make target. This target specifies which sensor node platform to use as a default if no other arguments are specified. YETI does not offer a hard coded list of targets but queries the TinyOS make system for supported devices. This guarantees that for each installation of the TinyOS system all supported node platforms are available to the user.

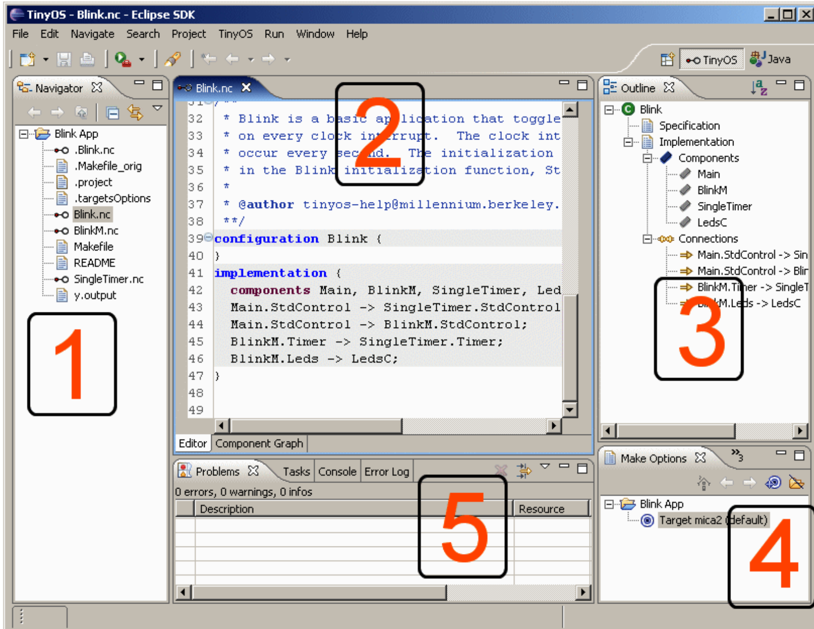

Figure 12.1: Screenshot of YETI. 1: Navigator listing all files of the project. 2: Main window showing the editor with the current file or the application graph. 3: Outline of the open file showing its structure. 4: Make option window containing predefined make targets. 5: Multi purpose panel hosting various features such as problem view, console and TinyOS search.

File Editing

For the development of applications YETI provides a customized editor supporting the nesC programming language. It features a correct syntax highlighting and incorporates various commodity functions known from other development environments. Its most important feature is definitively the real time spell checker. Syntactic and semantic errors are detected within a fraction of a second and are marked with a red X at the beginning of the line. Furthermore, an error message is generated in Eclipse's Problems log containing clickable links pointing to the corresponding location in the source code. For the most common problems such as missing semicolons the error messages also offer a suggestion on how to fix them (also see Section 12.4.2). Moreover, the editor contains a code completion function which can be used to create stubs for methods which have to be implemented in a file providing or using a specific interface.

Outline

For a better overview of the application YETI provides an *Outline* of the open file. This outline lists all *components*, *interfaces*, *modules* and *configurations* which are the building blocks of every TinyOS applications. With a simple click it is possible to open the declaration of an interface or to jump to a specific function within the open source file. Due to the lack of an explicit package structure within TinyOS multiple implementations of the same interface may be available. This is mostly the case if hardware specific features are used. For example, accessing hardware timers works differently on different processors and thus the *Timer* interface used in nearly all TinyOS applications has many custom implementations for the various sensor node platforms. YETI uses the currently chosen target platform to decide which of the various files to open. Consequently, the user always sees the implementation which will be used to compile the application.

TinyOS Specific Search

A TinyOS specific search function allows browsing through all available interfaces and to scan for modules implementing them. For this purpose the structure of the appropriate source files is parsed and evaluated. Several special search modes are available. *Interfaces*, *modules*, and *configurations*[2] can be listed, filtered and accessed from the search frame.

This feature is especially helpful since TinyOS uses a complex set of rules to decide which modules to include when compiling an application. YETI's

[2] In TinyOS *configurations* are used to combine several modules to an application or a subprogram.

search function follows all valid paths, including custom imports made by the user, to find files matching the entered search queries. This ensures that all valid files are found but no sources incompatible to the current make target are shown.

Compiling and Flashing

For compiling applications and flashing sensor nodes with the resulting binaries YETI relies on the TinyOS make system. However, users are no longer required to type in cryptic command line calls but a simple wizard helps setting up make options and stores them for later reuse. YETI automatically identifies all available target platforms for a given TinyOS installation and also examines further valid parameters such as possible extension boards. The identified options are displayed in a dialog and the user can create even complex make calls by means of simple point and click operations.

Feedback on the results of a call to the make system are printed to Eclipse's built in console. This user interface is not only easier to utilize but it also prevents the generation of invalid make calls. Furthermore, YETI allows batch execution of the make system simplifying the tedious process of reprogramming large numbers of nodes.

Application Graph

Another feature of the System plug-in is the *Application Graph*. This tool produces a graphical representation of the currently developed application and can be used to plot the relation between its modules. Exploiting the hierarchical structure of TinyOS modules the user can decide on the graph's level of abstraction by expanding or collapsing some of the elements. If required it is possible to expand the graph to show *all* modules forming the current application including the ones of the operating system. However, as can be seen in Figure 12.2 even simple programs such as the *Blink* demo application lead to complex graphs if fully expanded. Therefore, in most cases it is advisable to keep a certain level of abstraction to view the structure of a program.

12.3.2 TinyOS Environment Wrapper

The task of the TinyOS Environment Wrapper is to provide the System plug-in with a well-defined access to a TinyOS installation. This separation of development environment and TinyOS system has several advantages. First, it allows having several independent installations of TinyOS for different target platforms. This is desirable since in many cases the tool chain necessary to

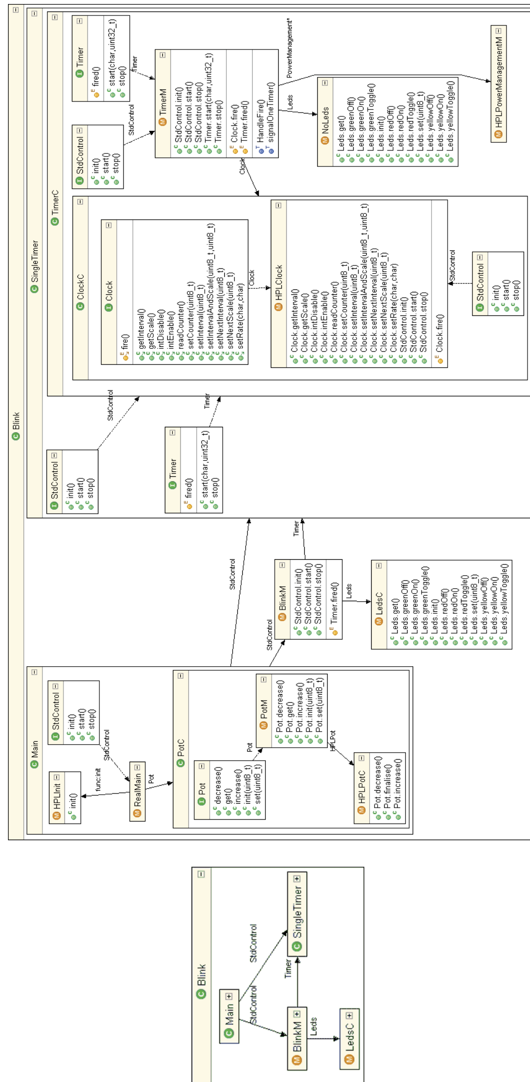

Figure 12.2: Graph of the Blink application at two different levels of abstraction

compile applications for one sensor node platform interferes with the tools for another one. This problem is one of the main reason why it is so tedious to test newly written applications on various nodes. With YETI it is a matter of one click to change between the different available TinyOS environments and to test the application on all available node types.

Another advantage of this separation is that hardware producers may provide their own TinyOS Environment wrappers. With such individually optimized TinyOS installations it can be ensured that the application developers work on a correctly configured environment. For the software developers this approach has the advantage that a newly installed environment will not interfere with or even destroy existing TinyOS installations as it is sometimes the case without YETI.

The only drawback of this approach is the increased hard disk space necessary to install several independent TinyOS environments. However, with the ever-growing hard disk sizes it should not be a problem to have several installations with a total size of one to two gigabytes.

In its initial release, YETI provided three different TinyOS Environment Wrappers. The first one contained a full installation of the current TinyOS 1.1.15 release. This wrapper was the best choice for most developers as it provided a stable environment with support for various sensor node platforms. Similarly, the second wrapper provided an installation optimized for the TinyNode 584 platform by Shockfish SA. These two wrappers provided an easy way to set up a new development environment with a preconfigured toolchain to build and flash the applications. The third wrapper was an "empty" skeleton wrapper which allowed the connection of YETI to an existing TinyOS installations. This wrapper was designed for experienced users with existing TinyOS installations who wanted to keep their individual setups.

In the mean time the installation process of the TinyOS sources and necessary tool chains has become much easier and more reliable. We have therefore decided to discontinue supporting our preconfigured installations and the current release of YETI only provides skeleton wrappers for TinyOS 1.x and 2.x.

12.4 Code Analysis

Most features such as the "Code Outline" or the spell checker require a syntactic and semantic understanding of the application which can only be achieved by scanning and parsing the source code. These operations need to be executed nearly in real time since users are unwilling to wait for several seconds before a new input is validated. At the same time the results have

to be correct or the development environment produces false alerts, making it mostly useless to the developer.

12.4.1 Scanner and Parser

The analysis of source code is traditionally split in three phases: lexical, syntactic, and semantic analysis. In the phase of the lexical analysis the nesC source files are tokenized. A token is defined as a sequence of logically connected items building atomic structures of the programming language. This includes keywords such as "module" or "implementation" and also strings, numbers, and type names. Tokens are created by comparing the source code to predefined patterns. The tool executing this lexical analysis is called *Scanner* or *Lexer*.

The goal of the syntactic analysis is to group the individual tokens and to validate their correctness according to a given grammar of a programming language. As a result of this analysis a syntax tree is built on which the semantic analysis is executed. In this last step unreasonable code which is syntactically correct is identified.

YETI contains a custom scanner and parser which were realized using *JFlex* [23] and *jay* [50], Java implementations of the well known tools *Lex* and *YACC*. Figure 12.3 shows a schematic representation of the internal interconnections between the parser and the visual tools of the development environment.

For the syntactical analysis a jay specification file was written, based on the nesC language definition found in [15]. With this specification as an input jay was used to create a finite state machine implementing a nesC parser. As can be expected this process was not straight forward. Starting with a YACC specification file for ANSI C the production rules were adapted to model the nesC programming language. Unfortunately, the resulting grammar was highly ambiguous leading to various *shift/reduce* and *reduce/reduce* conflicts. These problems had to be resolved by major reordering of parser rules.

Another problem arose from jay's limitation to create only a standard LR(1)[3] parser: Due to a conflict between identifiers and typedef-names, C and thus also its derivate nesC are not LR(n) compliant [33]. To avoid this problem extensions to the grammar were necessary.

Finally, nesC also supports individual name spaces for *configurations* and *interfaces*. Since these constructs are not known in pure ANSI C, the grammar had to be extended to consider these additional name spaces.

[3]LR indicates that rules are executed from left to right. The number in brackets specifies the number of tokens the parser can look ahead to optimize its decisions;

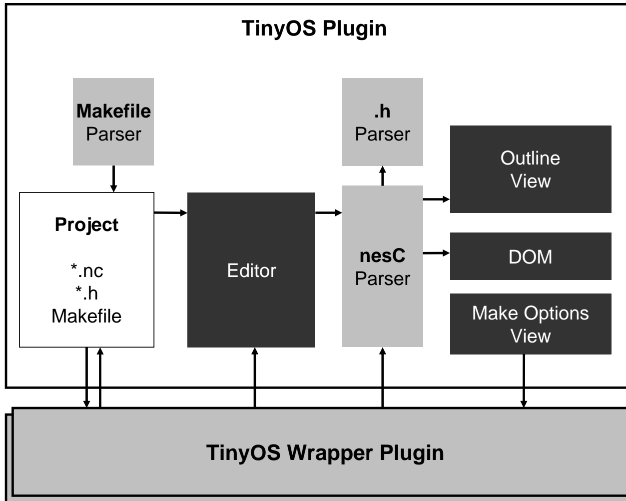

Figure 12.3: Internal configuration of YETI's components

12.4.2 Extending the Parser

Human readable error reports are crucial for any development environment. A simple output saying "Syntax Error" is not really helpful to any developer. What we want is an expressive report about the location and the nature of the problem. Jay already produces quite precise error statements but it is possible to further improve them. YETI provides a powerful mechanism to extend the parser's error messages by feeding it with specially prepared files.

To illustrate the process of adding a new error message to the parser Listing 12.1 shows an adapted configuration file of the *Blink* demo application. This version of the file differs from the original in that on line 1 an error message was added. Furthermore, on line 5 the character '>' was removed.

This file is now processed by a special tool included in YETI and the parser will analyze its content. The first line is stored as an error report but otherwise the parser ignores it completely. After parsing the rest of the file the error on line 5 is detected. The parser now stores the new custom error message in a consistent hashtable using a combination of its current internal state and the next expected token as a key. Next time the parser encounters the same problem it will check the hashtable for a custom error message. If the table contains an entry for the current parser state the stored message is

displayed. If there is no custom message known, the default output of jay is shown.

```
1  :::Wiring symbol '-'unknown, use '<-' or '->'
2  configuration Blink{}
3  implementation {
4   components Main, BlinkM, SingleTimer, LedsC;
5   Main.StdControl - SingleTimer.StdControl;
6   Main.StdControl -> BlinkM.StdControl;
7   BlinkM.Timer -> SingleTimer.Timer;
8   BlinkM.Leds -> LedsC.Leds;
9  }
```

Listing 12.1: Sample file teaching the parser a new error message if a '>' is missing in the wiring.

12.5 Related Work

To the best of our knowledge there are only two other, discontinued projects aiming at providing a development environment for TinyOS. Like YETI both of them are realized as Eclipse plug-ins but they differ in various respects.

The first tool is *TinyOS IDE* [57] by Richard Tynan which was the first publicly available TinyOS development environment. TinyOS IDE provides little advantages over using an advanced text editor and a shell. It provides syntax highlighting for nesC files and the option to compile applications from within Eclipse. However, to enable the compile function, it is necessary to have a preinstalled working TinyOS installation. Also the TinyOS specific environment variables need to be defined system wide or the tool cannot find the compiler. Similarly to YETI, TinyOS IDE allows to compose make calls by selecting the various options from a dialog. TinyOS IDE does not generate this dialog automatically but simply loads the information from a handwritten configuration file. The displayed make options are not guaranteed to be reasonable and if a parameter is required which is not available in the default menu the configuration file must be adapted. TinyOS IDE does not provide a spell checker but after building an application compiler errors are made available in Eclipse's error log.

The second tool is called *TinyDT* [46] and is developed at Vanderbilt University. TinyDT also provides a custom perspective within Eclipse and has an inbuilt TinyOS parser. Thus, TinyDT also provides a spell checker, an outline of the open file, and code completion for interface members. The parsers of TinyDT and YETI differ in one important aspect: While our parser is optimized for fast execution at the cost of some imprecision when

handling preprocessor statements, the parser of TinyDT is designed to be completely accurate. The drawback of this solution is a slow response time of the system. Even a change of one character in the source code takes several seconds before the file is revalidated and potential errors are detected.

Like TinyOS IDE also TinyDT requires a preinstalled TinyOS environment. The user needs to specify where to find the compilers for the different target platforms and the bash executable. TinyDT does not detect available node platforms but only supports nodes of the mica family, telos, telosb, and the tmote.

Chapter 13

Conclusion

In this thesis we investigated the development of applications for sensor networks with ultra-low power consumption. We thereby worked on two orthogonal axes of the problem. On the one hand, with Dozer, we have developed one of the most energy efficient data aggregation systems for wireless sensor networks.

Dozer started out as a joint venture with our industrial partner Shockfish SA and initially we did not expect it to turn into the almost three years long project it finally became. The first plan was to design and build a prototype and then leave the rest of the development to Shockfish. As planned, within a few weeks we had a first running release of the communication stack. However, we quickly learned why so many sensor network systems in the end did not perform as expected or even failed completely once deployed in the field: The devil was in the details. Even minor problems in the code could have disastrous impact on the performance of the system under some specific, rare conditions. As we believed in the potential of our algorithms we set out to understand and fix these problems. However, unlike planned, we could not outsource debugging to our industrial partner due to efficiency reasons.

On a personal level, to go through the full development cycle up to a market-ready sensor network solution we also had to extend our own horizon. At the beginning of the project we saw ourselves mainly as experts for distributed systems with a focus on communication protocols. With this background we were able to design the Dozer algorithms but for the implementation in TinyOS this background was not sufficient. For example, during the first development phase we visually controlled the timings of the communication stack by examining different blinking patterns of the LEDs on the sensor nodes. This primitive mode of observation worked well for timings

down to a couple of tenths of a second. However, from there on we needed new tools such as an oscilloscope to measure Dozer's performance. Similarly, we had to accept that for programming sensor networks it is unavoidable to learn more about the operating system and underlying hardware. Dependencies between hardware interrupts leading to unexpected side effects could often only be understood by going through the specifications of the hardware and the corresponding driver implementations. That is, we had to increase our knowledge in hardware related programming to understand the behavior of our own system.

The most important lesson we learned from this experience is that there is a duality in requirements to build a complex sensor network system. On the one hand a developer must have strong skills in distributed systems to foresee bottlenecks and pitfalls in the system. On the other hand, a sound knowledge of the employed platform and its limitations is required to write code capable of coping with the hassles it will face under real-life conditions. With Dozer we have proven that combining theoretical studies with thorough engineering enables the construction of complex, yet stable sensor network applications. It is our hope that this success will encourage and inspire further cooperation between scientists from the systems and theory communities and result in new exciting applications.

As a second topic of this thesis we have worked on development support tools to speed-up the implementation of applications for sensor networks. Development tools for wireless sensor networks are still in a very early stage and writing programs is similar to what we used to see on personal computers in the early nineties. That is, command line tools and simple text editors are the main work utensils for TinyOS developers. From our experiences teaching students how to develop applications for distributed systems in general and wireless sensor networks in specific, we knew about the problem of how to get started with this topic. At the beginning of this thesis a newcomer easily needed a week to setup the development tools necessary to write an application in TinyOS, compile the code, and flash it on a sensor node.

This experience was the main reason why we decided to start our most ambitious project in this domain, the YETI development environment. The project is still ongoing and during its more than one and a half years of development the Eclipse plugin has evolved into a complex system. With the release of TinyOS 2.x a complete overhaul of the plugin became necessary to handle the new language options of nesC. In the course of this update several modifications were introduced reflecting the changing requirements of TinyOS developers. On the one hand, by now redundant features such as the preconfigured TinyOS installations were removed from the installation bundle. On the other hand, new convenience features such as an improved search function or faster access to platform specific file definitions were intro-

duced. Furthermore, there are ongoing related projects extending YETI with new functionality such as JTAG debugging or the integration of a TinyOS simulator. As the different components of the IDE may also be of interest to other projects in the domain of development tools it was recently decided to release the sources of YETI on our website.

YETI is the most advanced IDE for TinyOS applications offering developers a more convenient and less error prone way to write their programs. We strongly believe that tools like YETI are a key to success for sensor network applications. By making the first contact with TinyOS more user friendly, new developers from areas outside the current sensor network community are attracted and with them novel application ideas will emerge, helping sensor networks to achieve their long envisioned breakthrough.

Bibliography

[1] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel. PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, pages 265–276, San Francisco, CA, USA, Apr. 2009. ACM/IEEE.

[2] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, and L. Thiele. Prototyping wireless sensor network applications with BTnodes. In *Proc. 1st European Workshop on Sensor Networks (EWSN 2004)*, volume 2920 of *Lecture Notes in Computer Science*, pages 323–338. Springer, Berlin, Jan. 2004.

[3] T. Brooke and J. Burrell. From Ethnography to Design in a Vineyard. In *DUX '03: Proceedings of the 2003 conference on Designing for user experiences*, pages 1–4, New York, NY, USA, 2003. ACM Press.

[4] R. Cardell-Oliver, K. Smettem, M. Kranz, and K. Mayer. A Reactive Soil Moisture Sensor Network: Design and Field Evaluation. *Int. Journal of Distributed Sensor Networks*, 1(2):149–162, 2005.

[5] Crossbow Technology. MICA2 Wireless Measurement System. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf`.

[6] CTI/KTI. http://www.bbt.admin.ch/kti/.

[7] A. Dunkels, J. Alonso, T. Voigt, H. Ritter, and J. Schiller. *Connecting Wireless Sensornets with TCP/IP Networks*, pages 583–594. Springer, Berlin.

[8] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of

wsns. In *European Conference on Wireless Sensor Networks (EWSN)*, 2007.

[9] A. El-Hoiydi and J.-D. Decotignie. WiseMAC: An Ultra Low Power MAC Protocol for Multi-hop Wireless Sensor Networks. In *Int. Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS)*, 2004.

[10] K. Fall. NS Notes and Documentation. *The VINT Project*, 2000.

[11] R. Fan and N. Lynch. Gradient clock synchronization. *Distrib. Comput.*, 18(4):255–266, 2006.

[12] R. Flury. Routing on the Geometry of Wireless Ad Hoc Networks. In *PhD Thesis, ETH Zurich, Diss. ETH No. 18573*, September 2009.

[13] J. Flynn, H. Tewari, and D. O'Mahony. Jemu: A real time emulation system for mobile ad hoc networks. In *Proceedings of the Firsit Joint IEI/IEE Symposium on Telecommunications Systems Research, Dublin, Ireland*, November 2001.

[14] S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149, New York, NY, USA, 2003. ACM.

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[16] M. Günes, B. Blywis, and J. Schiller. A hybrid Testbed for long-term Wireless Sensor Netowrk Studies. In *International Workshop on Sensor Network Engineering (IWSNE'08)*, 2008.

[17] H. Dubois-Ferrier and R. Meier and L. Fabre and P. Metrailler. TinyNode: a comprehensive platform for wireless sensor network applications. In *Int. Conference on Information Processing in Sensor Networks (IPSN)*, 2006.

[18] A. Heybey. The network simulator. Technical report, MIT, September 1990.

[19] B. Hohlt and E. Brewer. Network Power Scheduling for TinyOS Applications. In *EEE Int. Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2006.

[20] B. Hohlt, L. Doherty, and E. Brewer. Flexible Power Scheduling for Sensor Networks. In *Int. Conference on Information Processing in Sensor Networks (IPSN)*, 2004.

[21] J. W. Hui and D. E. Culler. Ip is dead, long live ip for wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 15–28, New York, NY, USA, 2008. ACM.

[22] R. Kehrer. Link Layer Measurements in Wireless Sensor Networks. 2005.

[23] G. Klein. JFlex. `http://jflex.de`.

[24] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, 2006.

[25] C. Lenzen, T. Locher, P. Sommer, and R. Wattenhofer. Clock Synchronization: Open Problems in Theory and Practice. In *36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Spindleruv Mlyn, Czech Republic*, January 2010.

[26] C. Lenzen, T. Locher, and R. Wattenhofer. Tight Bounds for Clock Synchronization. In *28th ACM Symposium on Principles of Distributed Computing (PODC), Calgary, Canada*, August 2009.

[27] C. Lenzen, P. Sommer, and R. Wattenhofer. Optimal Clock Synchronization in Networks. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys), Berkeley, California, USA*, November 2009.

[28] G. Lu, B. Krishnamachari, and C. Raghavendra. An Adaptive Energy-Efficient and Low-Latency MAC for Data Gathering in Sensor Networks. In *Int. Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (WMAN)*, 2004.

[29] H. R. Lundgren, D. Lundberg, E. Nordstrm, and C. F. Tschudin. A Large-scale Testbed for Reproducible Ad hoc Protocol Evaluations. In *Proceedings of IEEE WCNC*, 2002.

[30] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Systems*, 30(1):122–173, 2005.

[31] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM Int. Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[32] K. Martinez, P. Padhy, A. Elsaify, G. Zou, A. Riddoch, J. Hart, and H. Ong. Deploying a Sensor Network in an Extreme Environment. In *IEEE Int. Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, 2006.

[33] W. M. McKeeman. Resolving Typedefs in a Multipass C Compiler, March 1991.

[34] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 packets over IEEE 802.15.4 networks. In *RFC 4944 (Proposed Standard)*, 2007.

[35] moteiv. Tmote Sky. `http://www.moteiv.com/products-tmotesky.php`.

[36] R. Musaloiu-E., C.-J. M. Liang, and A. Terzis. Koala: Ultra-low power data retrieval in wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 421–432, Washington, DC, USA, 2008. IEEE Computer Society.

[37] T. Naumowicz, R. Freeman, A. Heil, M. Calsyn, E. Hellmich, A. Brändle, T. Guilford, and J. Schiller. Autonomous monitoring of vulnerable habitats using a wireless sensor network. In *REALWSN '08: Proceedings of the workshop on Real-world wireless sensor networks*, pages 51–55, New York, NY, USA, 2008. ACM.

[38] D. Networks. http://www.dustnetworks.com/.

[39] L. Ni and P. Zheng. EMPOWER: A Network Emulator for Wireline and Wireless Networks, 2003.

[40] R. Panta, I. Khalil, and S. Bagchi. Stream: Low overhead wireless reprogramming for sensor networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 928–936, May 2007.

[41] K. Pister and L. Doherty. TSMP: Time synchronized mesh protocol. In *Parallel and Distributed Computer and Systems*, 2008.

[42] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Int. Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[43] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, 2000.

[44] J. M. Rabaey, M. J. Ammer, J. L. da Silva, D. Patel, and S. Roundy. PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking. *Computer*, 33(7):42–48, 2000.

[45] V. Rajendran, J. Garcia-Luna-Aceves, and K. Obraczka. Energy-Efficient, Application-Aware Medium Access for Sensor Networks. In *IEEE Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, 2005.

[46] J. Sallai, G. Balogh, and S. Dora. TinyDT. `http://www.tinydt.net`.

[47] J. Schiller, A. Liers, H. Ritter, R. Winter, and T. Voigt. Scatterweb - low power sensor nodes and energy aware routing. In *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 286c–286c, Jan. 2005.

[48] T. Schmid, H. Dubois-Ferrière, and M. Vetterli. SensorScope: Experiences with a Wireless Building Monitoring Sensor Network. In *Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2005.

[49] L. Schor, P. Sommer, and R. Wattenhofer. Towards a Zero-Configuration Wireless Sensor Network Architecture for Smart Buildings. In *First ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings (BuildSys), Berkeley, California, USA*, November 2009.

[50] A.-T. Schreiber and B. Kuehl. jay. `http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay`.

[51] Shockfish SA. http://www.tinynode.com.

[52] R. Simon, L. Huang, E. Farrugia, and S. Setia. Using multiple communication channels for efficient data dissemination in wireless sensor networks. In *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*, pages 10 pp.–439, Nov. 2005.

[53] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM.

[54] R. Szewczyk, J. Polastre, A. M. Mainwaring, and D. E. Culler. Lessons from a sensor network expedition. In *EWSN*, pages 307–322, 2004.

[55] The Eclipse Project. http://www.eclipse.org.

[56] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Int. Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[57] R. Tynan. TinyOS IDE. `http://tinyoside.ucd.ie`.

[58] T. van Dam and K. Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Int. Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[59] P. von Rickenbach and R. Wattenhofer. Decoding Code on a Sensor Node. In *4th International Conference on Distributed Computing in Sensor Systems (DCOSS), Santorini Island, Greece*, June 2008.

[60] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68, Piscataway, NJ, USA, 2005. IEEE Press.

[61] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.

[62] Wireless HART. http://www.hartcomm2.org.

[63] W. Ye, J. S. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[64] W. Ye, F. Silva, and J. S. Heidemann. Ultra-Low Duty Cycle MAC with Scheduled Channel Polling. In *Int. Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.

[65] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

Curriculum Vitae

May 2, 1978	Born in Basel, Switzerland
1985–1997	Primary and high schools in Basel, Switzerland
1997–2004	Studies in computer science, ETH Zurich, Switzerland
April 2004	Diploma in computer science, ETH Zurich, Switzerland
2004–2010	Ph.D. student, research and teaching assistant, Distributed Computing Group, Prof. Roger Wattenhofer, ETH Zurich, Switzerland
April 2010	PhD degree, Distributed Computing Group, ETH Zurich, Switzerland Advisor: Prof. Roger Wattenhofer Co-examiner: Prof. Jochen Schiller, FU Berlin, Germany