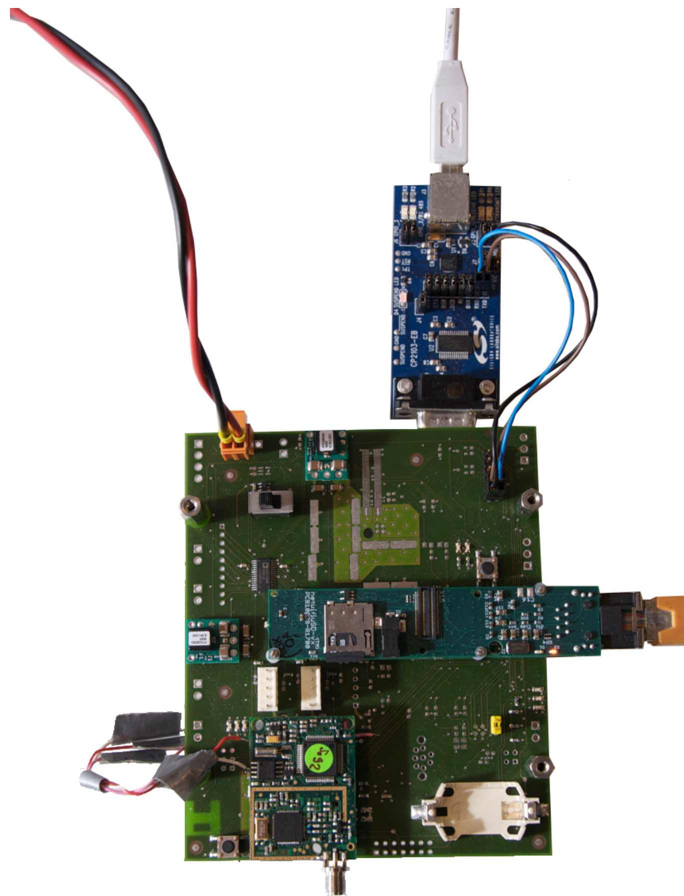
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Prof. Dr. L. Thiele
Fall term 2008

MASTER THESIS

Wireless Sensor Network Testbed 2.0: A New Service Oriented Architecture



Author:
Christoph Walser

Supervisors:
Matthias Woehrle
Andreas Meier

Zusammenfassung

Drahtlose Sensor Netzwerke (WSN) werden vielfach in schwer zugänglichen Gebieten installiert was es schwierig macht, solche Netzwerke für Updates und zur Fehlerbehebung zu erreichen. Es ist daher wichtig, dass ein WSN korrekt funktioniert bevor es installiert wird. Um möglichst grosse Korrektheit erreichen zu können werden Validierungs-Werkzeuge benötigt. Prüfstände (Testbeds) haben dabei ihre Nützlichkeit wiederholt bewiesen da mit ihrer Hilfe komplizierte Details von WSNs und deren Interaktion mit der Umgebung aufgespürt werden können.

Ein bestehendes Testbed nutzt eingebettete Knoten welche mit den Sensor-Knoten verbunden sind. Diese Observer sind in der Lage, die Sensor-Knoten zu überwachen und zu steuern. Ein zweites, unterstützendes Netzwerk dient der Kommunikation zwischen den Observern und einem Server. Einzelne Sensor-Knoten können mit traditionellen Labormessgeräten für eingebettete Systeme, wie zum Beispiel Energiemessgerät oder Logikanalysator, instrumentiert werden. Viele Details von WSN können aber nur erfasst werden, wenn viele Knoten gleichzeitig überwacht werden. Deshalb wurde ein neues Testbed konzipiert, in welchem jeder Observer-Knoten in der Lage ist, Messungen ähnlich jenen mit Labormessgeräten vornehmen zu können. Dies resultiert in verteilten Messmöglichkeiten über das gesamte Testbed.

In dieser Masterarbeit wird ein Service für das neue Testbed konzipiert, implementiert und evaluiert welcher es ermöglicht, Pins des Observer-Prozessors zeitgesteuert zu setzen und diese Änderungen auf dem Sensor-Knoten zu detektieren. Anhand dieses beispielhaften Services wird der neue Observer evaluiert.
Vorausetzung für das Funktionieren des Services auf dem Observer ist ein Speichersystem für anfallende Messdaten sowie eine genaue Zeitbasis für die Abarbeitung und Protokollierung der Pin-Änderungen. Zur Datenspeicherung wird ein Datenbank-System ausgewählt und anhand eines entwickelten Datenbank-Modells implementiert. Um eine Zeitbasis zwischen allen Observern zu schaffen, wird ein Zeitsynchronisations-Protokoll implementiert und so konfiguriert, dass alle Observer synchronisiert werden.

Die Evaluation der Observer-Hardware und aller implementierter Software zeigt, dass das System geeignet ist um im neuen Testbed benutzt zu werden. Dies wiederum rechtfertigt die weitere Entwicklung des neuen Testbeds.

Abstract

Wireless sensor network applications are often deployed in remote areas. It is difficult to access them for updates and debugging purposes, hence it is critical for them to be correct at deployment time. In order to ensure correctness of the applications, validation tools are needed. Testbeds have proven to be valuable for validating wireless sensor networks because test executions on testbeds capture intricate details of device and environmental characteristics.

A current testbed uses embedded nodes, which are connected to the sensor nodes, as observers. The observers are able to monitor and control the sensor nodes. A backbone network is used for communication between the observers and a server. In the current testbed, only dedicated nodes can be instrumented with traditional embedded laboratory instruments, e.g. power profiling or logic analyzer that allow for a detailed analysis. However, many details of wireless sensor network applications are only revealed when doing such measurements on several sensor nodes simultaneously. Consequently, a new testbed has been designed. The idea is that each observer node includes means for detailed control and analysis such as stimulation and power analysis similar to embedded laboratory instruments thus allowing for distributed measurement rather than having dedicated equipment for just a few nodes.

In this master thesis, an exemplary testbed service is designed, implemented and evaluated which allows for setting pins on the processor of an observer and detecting the pin changes on a sensor node. Based on this service, the new observer is evaluated. The prerequisites for fully working services on the observer are a system to store measurement data and an accurate time base to be able to provide high-precision timestamps. For storing data, a database system is chosen and set up according to a developed database model. To provide a common time base on all observers, a time synchronization protocol is set up and configured to synchronize all observers.

The evaluation of the observer hardware and all implemented software shows that the system is well suited to be used in the new testbed. This justifies the further development of the testbed.

Preface

Many people contributed to the success of this master thesis. First, I want to thank Anna-Laura who supported me whenever I did not see the wood for the trees. Together with my two supervisors, Matthias and Andreas, I spent many hours discussing theoretical issues and problems of my thesis. They always had time for me and I very much appreciate this. Mustafa and Roman supported me whenever I had practical questions about the DSN, Linux, hardware components or lab equipment. Last but not least I want to thank my proof-readers Anna-Laura, Ruth, and Mel who did a great job when reading the report, looking for typos and grammatical mistakes.

Zürich, 7. April 2009

Christoph Walser

Contents

1 Introduction **1**
- 1.1 Wireless Sensor Networks . . . 1
- 1.2 Validation of Wireless Sensor Networks . . . 2
- 1.3 Testbeds . . . 4
  - 1.3.1 Deployment Support Network . . . 4
  - 1.3.2 MoteLab . . . 6
  - 1.3.3 TWIST . . . 6
  - 1.3.4 Comparison . . . 6
- 1.4 Problem Statement and Scope of this Thesis . . . 7
- 1.5 Chapter Overview . . . 7

2 Conceptual Design **9**
- 2.1 Time Synchronization . . . 11
  - 2.1.1 Design Requirements . . . 12
  - 2.1.2 Network Time Protocol . . . 12
  - 2.1.3 NTP Network Layout . . . 14
- 2.2 Services . . . 15
  - 2.2.1 Design Requirements . . . 15
  - 2.2.2 GPIO Monitor . . . 16
  - 2.2.3 GPIO Setting . . . 16
  - 2.2.4 Logging Service . . . 17
  - 2.2.5 Power Profiling . . . 18
  - 2.2.6 Further Services . . . 18
- 2.3 Data Storage . . . 18
  - 2.3.1 Design Requirements . . . 19
  - 2.3.2 Database Architecture . . . 19
  - 2.3.3 Database Model . . . 19

3 Implementation **23**
- 3.1 Data Storage . . . 23
  - 3.1.1 Database Software . . . 23
  - 3.1.2 Server Implementation . . . 27
  - 3.1.3 Observer Implementation . . . 28

3.2 GPIO Setting Service . . . 28
3.2.1 Overview . . . 28
3.2.2 API . . . 29
3.2.3 Kernel Module . . . 30
3.2.4 Database Daemon . . . 32

4 Service Evaluation 33
4.1 Pin Setting Variance . . . 35
4.1.1 Target Node Variance . . . 35
4.1.2 Observer Variance . . . 36
4.2 Pin Setting Offset . . . 37
4.2.1 Test Setup . . . 37
4.2.2 Test Results . . . 37
4.3 Time Synchronization Effects . . . 39
4.4 Synchronicity of Observers . . . 39
4.4.1 Test Setup . . . 41
4.4.2 Test Results . . . 42
4.5 Conclusions . . . 42

5 Conclusion and Future Work 45

A Database Schematics 51

B Task Description 53

C Work Schedule 59

List of Figures

1.1 Schematics of a wireless sensor network 1
1.2 Development steps of a wireless sensor application 2
1.3 Categorization of testing tools 3
1.4 Layout of the DSN testbed 4

2.1 Target-observer model 9
2.2 Implementation of target-observer model 10
2.3 Overview of target-observer interface 11
2.4 Clock model of observer 11
2.5 Time synchronization network layout 14
2.6 Overview of services 16
2.7 Typical wakeup time for a radio chip 17
2.8 Database model 20

3.1 Overview of GPIO setting service 29
3.2 Program flow of kernel module 30
3.3 Steps to setup an OS timer 31

4.1 Instrumentation of testcase 34
4.2 Drift of signals with different frequencies 35
4.3 Offset of pin setting on observer 38
4.4 Effects of time synchronization 40
4.5 Time synchronization of chrony at system startup 41

List of Tables

1.1 Differences between simulations and testbeds 3
1.2 Overview of testbeds 5

3.1 Runtime of n insert statements on the database 25
3.2 Runtime for insertion of 5000 samples 25
3.3 Time to insert one sample 26
3.4 Memory requirements for database 26

4.1 Mean error of target node when detecting pin changes. 35
4.2 Mean error of observer at 0.5Hz 36
4.3 Mean error of observer at 50Hz 36
4.4 Time difference between two observers when setting pins at synchronized time intervals. 42
4.5 Time difference between two observers when monitoring pins 42
4.6 Worst case variance of GPIO setting service 43

Chapter 1

Introduction

1.1 Wireless Sensor Networks

Wireless sensor nodes are embedded systems typically equipped with a micro controller, memory, radio, energy supply and one or several sensors. The purpose of these nodes can be among others to do measurements related to environmental conditions in high-alpine regions. The nodes are therefor deployed in a sector of interest on a mountain. As not all sensor nodes have a direct connection to the sink node, which acts as a base station, the measurement data is forwarded through an ad-hoc multi-hop network from node to node until it eventually reaches the sink. This can be seen in Figure 1.1 where sensor node x does not have a direct connection to the base station but forwards its data to the sink over sensor node y. The base station forwards the data from all nodes to an external location for further processing.

As the nodes are normally powered with a battery, energy efficiency is of utmost importance. Software running on sensor nodes needs to be optimized, e.g. the network stack needs to be adapted as was done in [2].
Further constraints for sensor nodes, besides limited energy supply, are communication range, processing power, and memory size.

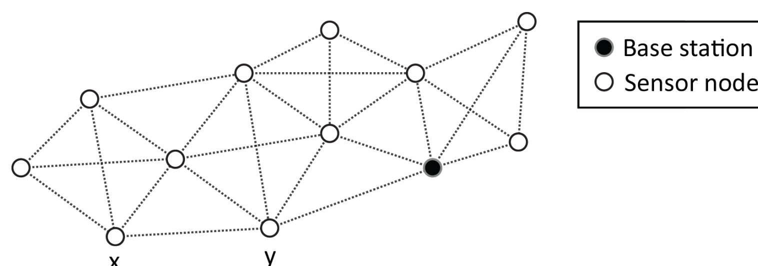


Figure 1.1: Schematics of a wireless sensor network[1]. The sensor nodes form an ad-hoc multi-hop network and forward messages to the base station.

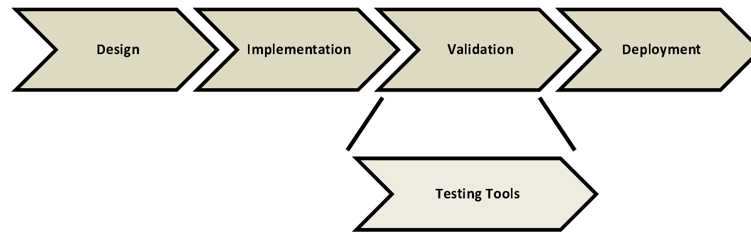


Figure 1.2: Phases in the development of a wireless sensor network application. For the validation phase, tools are needed for testing the WSN.

Wireless sensor networks (WSN) are an emerging field in information technology research. Since the first (wired) sensor networks in the 1950s (e.g. SOSUS[3]) with a military background, sensor networks have become an important field of studies with many application areas. Examples of areas and typical applications are:

- Environment:
  - Bird habitat monitoring: Great Duck Island (2002) [4]
  - Permafrost research: PermaSense (2006) [5]
  - Microclimate monitoring: Redwood Tree (2005) [6]
  - Precision agriculture: LOFAR-agro (2005) [7]
  - Volcanology: Monitoring of volcanic activity (2005) [8]
- Other:
  - Health monitoring: Medical Body-Sensor Networks (∼2003) [9]
  - Intrusion detection: A line in the sand (2003) [10]
  - Oil pipeline infrastructure monitoring: Pipenet (2004) [11]

A good overview of applications of WSNs is given by Römer et al. [12].

1.2 Validation of Wireless Sensor Networks

The realization of a WSN from the first idea to the final deployment can be distinguished into phases according to Figure 1.2. The validation of the WSN is the final step before deployment. According to Figure 1.3 there are basically two methods for testing a WSN: simulation and testing on real devices.
Real device testing can be further distinguished into small scale testing with a few nodes on the desktop of the WSN developer, and testbed testing with numerous nodes in a realistic environment. As desktop testing allows only for testing of a very limited number of nodes, testbeds are needed for extensive testing of networking

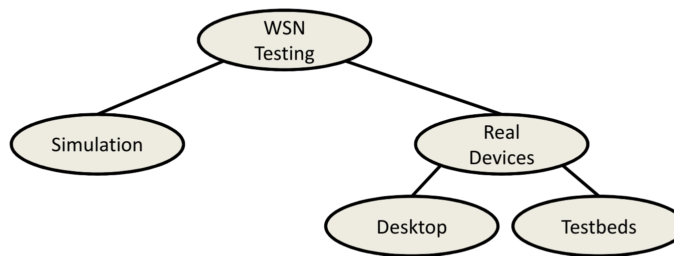


Figure 1.3: Most available testing tools can be categorized into simulation and real device testing tools.

	Simulation	Testbed
Pro	Hardware cost Visibility Controllability Repeatability Speed	Realistic environment Data quality
Contra	Simplistic/inaccurate model Data quality	Visibility Intrusiveness/probing effect Hardware cost Controllability Repeatability Speed

Table 1.1: Differences between simulations and testbeds. Advantages of one tool are generally disadvantages of the other and vice versa.

issues or the behavior of distributed algorithms.
Simulation tools and testbeds have different advantages and disadvantages and are therefor commonly used complementary during the validation phase of a WSN. The main differences between simulations and testbeds are listed in Table 1.1. Simulation software such as ns-2 [13], GloMoSim[14] or TOSSIM [15] help testing program flows. A large number of nodes can be simulated on a single computer, resulting in very low hardware cost for this type of testing. Another important advantage of simulation tools is their good visibility. The visibility defines how good the program state of a node can be inspected. Simulations have a very good visibility as almost every state of the node can be inspected at any time. However, the quality of simulation data can be poor. One problem is that most simulation tools have insufficient models for communication channels whereas testbeds use real devices and real communication channels making them the weapon of choice e.g. for testing communication issues. Testbeds provide less visibility but higher data quality. As a simulation can be paused at any time and the state of simulated nodes can be set or read easily, simulation tools provide a high controllability and make it easy to repeat test scenarios multiple times by putting the nodes into a state of interest and

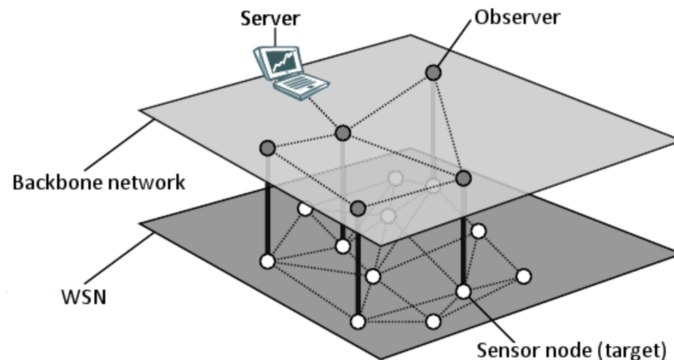


Figure 1.4: Layout of DSN.

letting the simulation run. Testbeds on the other side do not provide this ability as they run continuosly.
Summing up, it can be stated that the main advantage of simulations is the high visibility at low cost which they provide. On the other hand, testbeds are a lot more realistic and the test data is generally of better quality.

1.3 Testbeds

Commonly referred testbeds are DSN [16], MoteLab [17], TWIST [18], and Kansei [19]. A list of the functionality provided by DSN, MoteLab and TWIST can be found in Table 1.2. Kansei was not included in this survey as its design and purpose differ significantly from the other testbeds.

1.3.1 Deployment Support Network

The deployment support network (DSN) [16] has a number of features that distinguish it from other testbeds [17, 18]. DSN can read and write data to the target node over the serial interface. Most other testbeds can only read data from target nodes and cannot pass commands to them.
DSN is built in a modular way as depicted in Figure 1.4. The target node under test is connected over a short wire to a so called observer (see Chapter 2 for detailed discussion) which runs the testing software, collects all measurement data and forwards it to the testbed server. The observer nodes form a Bluetooth scatter network within the DSN making the testbed mobile for outdoor testing or quick change of density patterns as the nodes can be relocated without having to rewire everything.
On the other hand, the Bluetooth scatternet, which builds the backbone network of DSN, suffers from capacity problems when the testbed is running tests that involve many nodes. The observer nodes [20] of the DSN suffer from three problems: first the Bluetooth radio is not able to send more than 15 packets per second which is

Testbed		DSN	MoteLab	TWIST
Services				
	Automation	x	x	
	Realtime data analysis		x	
	Data logging	x	x	x
	Event logging	x	x	x
	Remote commands	x	x	x
	Remote programming	x	x	x
	Power measurement	x	x	
Architecture				
	Flat	x	x	x
	Hierarchical			x
	Partitioning		x	x
	Indoor use	x	x	x
	Outdoor use	x		
Target nodes				
	Telos family	x	x	x
	Mica family	x	x	
	TinyNode	x		
	Others	x		x
	Number of nodes	40	190	90
Backbone				
	Ethernet		x	x
	Bluetooth	x		
	USB			x
Server services				
	Database	x	x	x
	Job scheduling		x	
	User interface	x	x	x
Specialties				
	Observer	x		
	Remote power control			x

Table 1.2: Overview of functionality of the most commonly referred testbeds.

not fast enough for transmitting measurement data at high rates. Second, they are not fast enough to process data at high rates and third they do not have the capability to store data persistently on the node. Furthermore, the DSN power profiling functionality turned out not to be satisfactory as the ADC sampling rate is too low to be able to profile, for example, radio uptime (which is in the range of a few milliseconds).

1.3.2 MoteLab

MoteLab [17] is a large testbed developed at Harvard University in 2005. It consists of up to 190 nodes which are permanently deployed indoors - there is no possibility for outdoor testing as with DSN. The target nodes of MoteLab are connected to a central server over Ethernet interface boards which is another difference to DSN - there is no observer involved. This results in a high intrusiveness on the device under tests as all the instrumentation for the different measurements has to be made on the target nodes themselves. Power profiling is done as on the DSN by attaching a network-connected digital multimeter to a single node.
A server service that is not present on the other testbeds is the job scheduling functionality. Test runs are defined as jobs and scheduled to run in a time window assigned to the test. This allows for arbitration among the users of the testbed. However, the disadvantage is that the users have no more control over the test run once it is scheduled. They can therefore not access nodes or change parameters during their time window.

1.3.3 TWIST

TWIST [18] is a testbed developed in 2006 at the Technical University of Berlin. It is deployed indoors only, using a hierarchical architecture of the backbone network consisting of USB hubs which connect to the target nodes and USB-to-Ethernet-Interfaces which connect the USB hubs to an Ethernet network. The testbed is controlled using a host computer as control station and an additional server for collecting measurement data. This architecture makes the backbone design complicated and injects problems due to incompatibilities of the different used technologies. Nevertheless, the architecture supports different hierarchical models for the target nodes and it allows for partitioning of the testbed.
Disadvantages of TWIST are the few services that are offered and the high intrusiveness of the instrumentation on the target node due to the missing observer (same as for MoteLab). A specialty of TWIST though is the ability to remotely power the nodes on and off. This opens the possibility of simulating node failures.

1.3.4 Comparison

A disadvantage of all presented testbeds is that there is no possibility for distributed measuring on all nodes concerning power measurements [21] and distributed logic analysis. The presented testbeds have single nodes that can be connected to a multimeter, logic analyzer or oscilloscope. The approach taken in the design of the testbed evaluated in this thesis is different. The idea of this new testbed is to provide power measuring and logic analysis on every node of the testbed opening new possibilities for testing WSNs.

1.4 Problem Statement and Scope of this Thesis

Based on the experiences made with the deployment support network DSN [16], a new testbed design has already been developed incorporating a more powerful observer. An observer platform was chosen and an interface hardware board was built to interconnect the observer with the target node.
Based on this preliminary work, the task for this master thesis is to design the services provided by the observer. Components needed for running the services are evaluated and implemented. An exemplary service is implemented and evaluated. Based on the results, the chosen observer hardware is evaluated regarding its suitability for the testbed.

1.5 Chapter Overview

The remainder of this thesis is devided into four parts. In Chapter 2, the conceptual design of the testbed and especially the observer, including all subcomponents of the observer, is discussed. Details of all implementations put into practice are given in Chapter 3. These implementations are evaluated and analyzed in Chapter 4. In Chapter 5, the final conclusions are derived and directions for future work are given.

Chapter 2

Conceptual Design

The new testbed has to overcome the deficiencies of the DSN (see Section 1.3.1). This is done by making the observer more powerful and changing the network technology from Bluetooth to Ethernet (both, wired and wireless ethernet are possible).

The new testbed makes extended use of the target-observer model (see Figure 2.1). In this model, an observer is connected over a wired interface to the device under test, the target node. The implementation of the target-observer model used for this thesis can be seen in Figure 2.2. The target node used is a Shockfish TinyNode 584 equipped with a TI MSP430 microcontroller. A Gumstix Verdex XL6P motherboard, equipped with a Marvell PXA270 XScale processor, 128MB RAM and 32MB flash memory plus an ethernet expansion board build the observer.
The connection interface provides connectivity and power supply for both target node and observer. Connectivity consists of the serial universal asynchronous receiver/transmitter (UART) interface and dedicated general purpose input/output (GPIO) lines which connect GPIOs of the observer processor (CPU) to GPIOs of the target microcontroller (MCU). An overview of the full-blown interface can be seen in Figure 2.3.
The testbed backbone network is based on wired ethernet, which is interference-

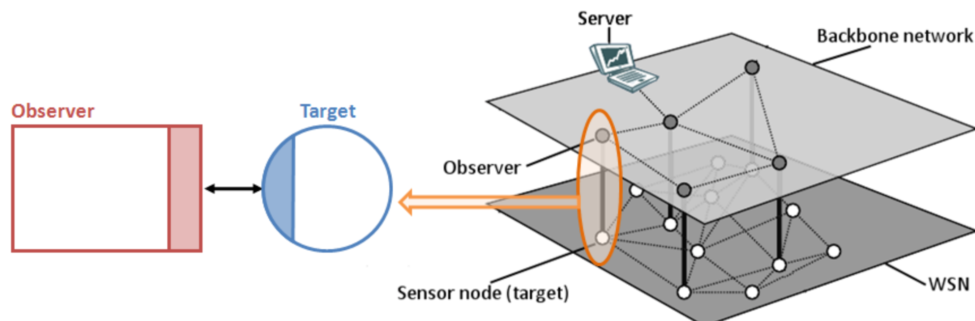


Figure 2.1: Target-observer model.

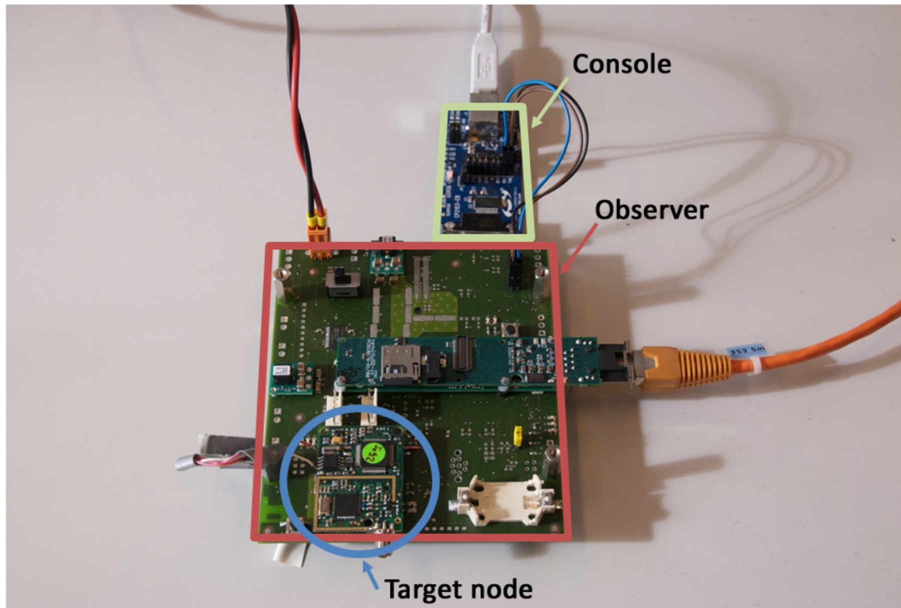


Figure 2.2: Implementation of target-observer model. The observer is accessed over a console attached to the board and powered by a 9V external power supply.

free against the wireless sensor network under test. If desired, the observer can be upgraded with a wireless ethernet expansion board to increase the mobility of the testbed.

Several requirements make the design of the new testbed challenging. The network which builds the backbone of the testbed has to be interference-free against the wireless sensor network. Of utmost importance is the reliability and robustness of the testbed. Reliability means that no data shall be lost due to system failure. Robustness means that the testbed has to run as it is supposed to - system crashes or otherwise unpredictable system behavior is undesired and is to be minimized.
The testbed has to be scalable from a few to a possibly large number of observers running at the same time. The observer is the base for all services and produces correlations between the different services (e.g. by providing consistent timestamps for all services running on the observer). Furthermore, the observer has to be extendable with more services if needed in the future.
The observers need to have a common timebase among each other and thus need to run on synchronized time. This makes timestamps comparable between all observers and allows for cross-analysis of target node behavior.

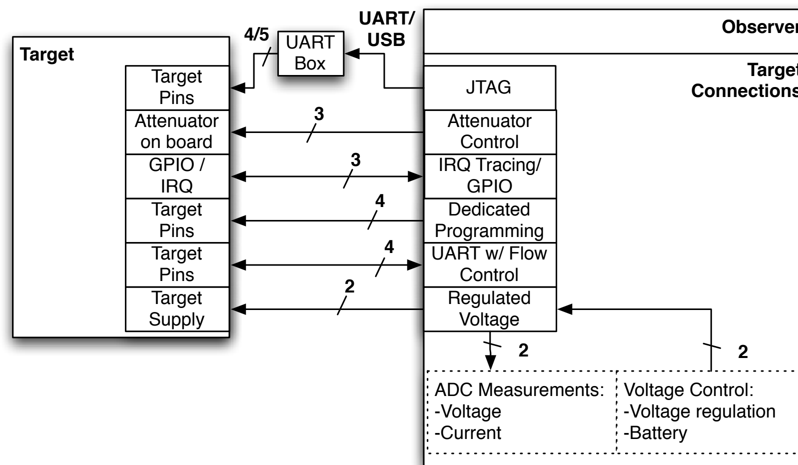


Figure 2.3: Overview of target-observer interface.

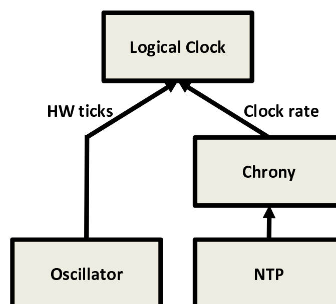


Figure 2.4: Clock model of observer.

2.1 Time Synchronization

Time synchronization serves two purposes in the testbed. First, it synchronizes all observers to run on a common timebase. This is needed to be able to compare time-stamps between observers to reconstruct for example network activity of the target nodes. Second, time synchronization is needed on the observers to compensate for their inaccurate clocks. Every observer is driven by a 13MHz oscillator which has an inaccuracy of ±50ppm[1] due to manufacturing tolerances, aging processes, and operating temperature sensitivity [22]. The oscillator basically updates a hardware clock which is implemented as a register in the observer. On top of the hardware clock is a logical clock which represents the time in the desired local format. The logical clock has thus an inaccuracy induced by the oscillator if not corrected. This is the point where time synchronization protocols such as NTP hop in and adjust the clock to run more precisely, see Figure 2.4.

[1]ppm = parts per million. +1ppm clock skew means that if the inexact clock runs for 1 second, in real-time it only advanced by 0.999999 seconds.

Time synchronization can thus be used to improve the accuracy of the internal clock of each observer and at the same time synchronize the time across all observers in the testbed. The time synchronization of the testbed can be seen as a side effect to the internal clock synchronization as it comes automatically when all observers synchronize their internal clocks based on data from the same time synchronization server.

2.1.1 Design Requirements

There are several requirements when using a time synchronization protocol:

- High precision. The time on each observer has to be as precise as possible, ideally in the low microseconds. This requirement holds even if the observer is not connected to the ethernet.
- High synchronicity. The observers need to have a common timebase. The difference of the internal clocks of all observers is to be in the low microseconds.
- Fast stabilization. The clock is to be synchronized as fast as possible after booting the observer.
- Continuous strictly monotone increase. The logical clock shall never run backwards, stand still or perform step changes in order to prevent unpredictable behavior of system components.

2.1.2 Network Time Protocol

The network time protocol (NTP) is a time synchronization tool which uses the Internet to gain access to high-precise time sources. NTP has a built-in hierarchy: a first level (the so called stratum 0) are atomic or radio clocks which provide the most accurate time signals possible. A second level (stratum 1) is connected to several of these servers and redistributes their time signals to the next level and so on. This system lets a user define its own time server and redistribute the time signal to a local network. With each additional stratum, the absolute accuracy of the respective time server decreases. For the testbed, the stratum of the local time synchronization server is not important as the local time of the observers does not have to be as precise as possible compared to real time but rather as precise as possible compared to the time of the local time server.
Each client synchronizes itself in configurable time intervals with a defined list of time servers and selects the server with the most accurate data [23]. From this data, the following values are calculated:

- Clock offset. The clock offset is the amount of time by which the local clock runs late or early.
- Roundtrip delay. The roundtrip delay is a value that represents the network delay to the chosen time server.

- Dispersion. Dispersion is the maximum error of the local clock relative to the reference clock.

On each client, an internal database is kept which stores this data over time. The clock synchronization becomes better the more data (that is the longer NTP is online) is available. NTP can achieve accuracies in the range of microseconds after a few hours runtime [23].

The logical clock of the clients can be adjusted in two different ways:

- Gradual phase adjustment. The frequency of the logical clock is adjusted to let it run either faster or slower.

- Step-change. If the offset of the clock is too large to become synchronized in useful time using gradual phase adjustment, a step-change is performed instantly to reduce the offset to zero. This mode hurts the requirement of a strictly monotonic increasing clock (see Section 2.1.1) and should thus be used with great care.

Gradual phase adjustments are done by adding more or less phase increments to the logical clock at periodic adjustment intervals.
Step-changes are performed by changing the register holding the value of the logical clock directly without doing any gradual phase adjustments.

2.1.2.1 Chrony

Chrony is a tool available for Unix systems to synchronize the logical clock using NTP. Chrony consists of two subprograms:

- Chronyd is a daemon that runs in the background of the system, communicates with the time servers and the clients and adjusts the local clock.

- Chronyc is the user interface to chronyd and provides means to configure and monitor the time synchronization.

Chrony supports both modes of NTP - gradual phase adjustments and step-changes. Chrony can be set manually to an offline-mode in which it continues to adjust the clock based on the time synchronization data last received from an online time server. It makes chrony thus an ideal tool for systems which are not guaranteed to always be online. As the testbed needs to be robust, chrony is ideal for time synchronization as in the case of a network failure it continues to operate.
With NTP and chrony it is possible to satisfy all requirements stated in Section 2.1.1.

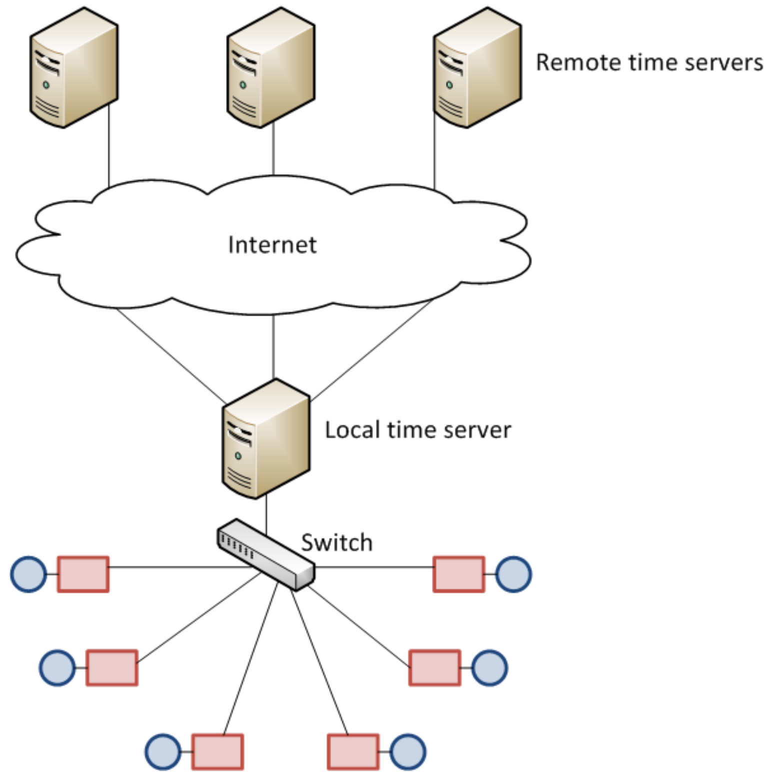


Figure 2.5: Time synchronization network layout.

2.1.3 NTP Network Layout

The time synchronization network layout chosen for the testbed can be seen in Figure 2.5. The central element is a local time server which synchronizes itself over the Internet with well-known time servers (such as for example swisstime.ethz.ch). It is possible to integrate the time synchronization server into the server which collects measurement data from the observers and holds the central database (see Section 2.3.3). The time synchronization server itself will not be perfectly accurate compared to real-time due to its low stratum but this is of no concern as it is not crucial for the observers to have a minimal offset compared to real-time but rather compared to each other.
In order to provide minimal path delay, all observers are connected through a dedicated ethernet switch to the time synchronization server. The observers are configured as client-only whereas the time synchronization server runs in master-slave-mode (master to the observers, slave to the servers in the next higher stratum). Chrony is configured to run in gradual phase adjustment mode only since a strictly monotonic increase of the observer clock is required by the testbed. A step change is performed however on system startup to bring the local clock to minimal offset as fast as possible.

2.2 Services

Service oriented architectures are characterized as systems which provide multiple services that can be combined in several ways to perform a multitude of tasks. Each component is a standalone implementation which fulfills a certain task. By using several services jointly, one can research a WSN taking a lot of different measurements into consideration and thereby discover correlations which would not be visible when running different measurements independently.

On a flexible testbed, a variety of different target nodes and software applications has to be tested. This leads to a number of different requirements for the testbed as not every application and/or every target node needs every testing possibility provided by the testbed. A way to provide the ability to use only a dedicated set of testing possibilities is the usage of a service oriented architecture. This lets the user choose what she wants to measure and which services should be turned off.

All services have a set of common features:

- Start and stop the service individually.
- When a service starts up, it shall indicate this to the server by setting an appropriate field in the observer database.
- Upon stopping of a service, the above mentioned field shall be reset and the service needs to be notified whether there is still data available on the observer for synchronizing.
- All errors that occur during operation of the service shall be reported to the database.

2.2.1 Design Requirements

Each service of the testbed allows for a specific measurement normally done with dedicated laboratory equipment. The integration of these measurement capabilities into the target and observer nodes leads to challenges regarding throughput, minimal intrusiveness and timing constraints. Services such as GPIO monitoring (see Section 2.2.2) or power profiling (see Section 2.2.5) have the potential to produce a lot of measurement data consuming CPU ressources for handling, memory space for storing, and bandwith for transferring the data to the server.
When running multiple services at the same time, the above stated problems also induce the challenge of timing constraints as the observer has to be fast enough to handle all work coming from the different services, to do appropriate timestamping, and not to loose data.
Two interfaces can be used for communication between the observer and the target as illustrated in Figure 2.6: the serial UART and dedicated GPIO lines. The GPIO monitoring and setting service use GPIO lines as they are fast and minimal

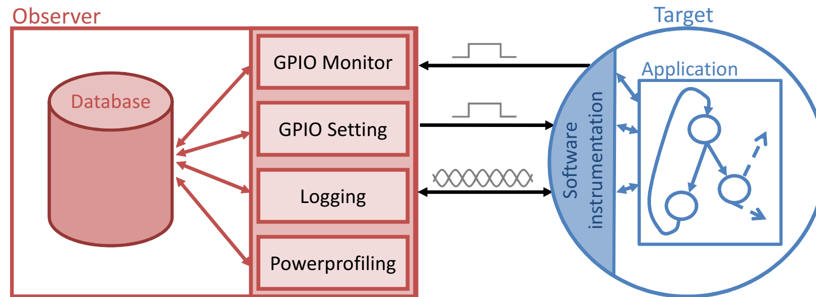


Figure 2.6: Overview of services and used communication lines between observer and target.

intrusive (it takes only a few cycles to read or write a GPIO in the target MCU). The logging service transports its log messages over the UART. The powerprofiling service hooks into the power supply of the target which is located on the observer and thus needs no software instrumentation on the target node.

2.2.2 GPIO Monitor

The GPIO monitoring service allows for monitoring the state of dedicated GPIO pins on the target node's microcontroller. This can be done by connecting one or multiple GPIO pins of the target node to GPIO pins of the observer node. The UART could be used as well for such a service but using GPIO pins has the advantage of minimal intrusiveness as setting a pin only needs a few cycles on the target node MCU. It is advantageous to use interrupt enabled GPIOs on the observer as this allows for accurate timestamping of the measured line state changes.
GPIO monitoring can be used for a variety of purposes when testing a WSN. One can be interested in state changes of the MCU's power mode, or the radio or software states to test a running application. State changes of the radio are perhaps the most interesting to look at because the radio switches quite fast from one state to another - a typical startup time for a radio (e.g. Chipcon CC2420) is around 3ms as can be seen in Figure 2.7. The individual phases of the radio are in the range of one to a few milliseconds. After the Nyquist-Shannon sampling theorem a sampling rate of at least 500μs is needed in order to detect these phases.
With the GPIO monitoring service at hand, one can instrument an application to set a particular pin combination for each state. By timestamping and logging this information on the observer, one can, for example, identify networking issues such as hidden terminal effects on the target nodes.

2.2.3 GPIO Setting

The GPIO setting service can be defined as the counterpart of the GPIO monitoring service which was presented in the previous section. The goal of this service is to be able to control the behavior of the target node by issuing commands over the

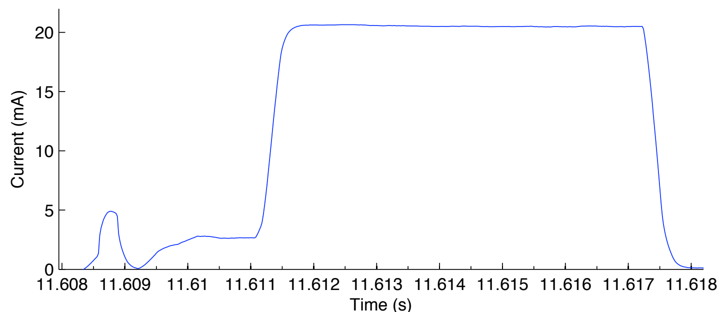


Figure 2.7: Typical wakeup time for a Chipcon CC2420 radio chip. [24]

GPIO pin interface of the microprocessor. One GPIO line can be set or cleared by the observer. The target starts some user defined action after receiving a signal from the observer.
The GPIO setting service can be used to start a memory dump of the target node's RAM which can then be examined and analyzed offline. This can be useful for debugging network issues. Another scenario imaginable with memory dumps is to upload RAM states taken earlier to the target nodes and flash them at a given time which can be used to preset the target nodes to a specific network state so one can, for example, examine effects that happen regularly after a certain time span without having to wait for this time to elapse.
One can also imagine to trigger a command to send data over the radio to multiple nodes at the same time. Using the GPIO monitoring service at the same time, collisions on the ether can be observed and thus the networking algorithms can be tested for effects of unwanted collisions.

2.2.4 Logging Service

Using the GPIO monitoring service one can access very little information as each monitored GPIO pin can only have two states - thus one can derive 1bit of information from every GPIO monitored. Sometimes it is easier to derive information form string-like messages as they can carry more information at the cost of lower throughput.
This functionality was already implemented in the first version of the testbed and proved to be a valuable tool for testing and debugging on the target node. As this service communicates by sending string-like messages over the UART interface to the observer, the service can never achieve the same low intrusiveness as the GPIO monitoring service because sending messages over the UART needs a lot more ressources on the target node than setting a GPIO pin. Nevertheless it is a very valuable functionality of the testbed when it comes to simple testing issues where one is not interested in complex relations between multiple nodes and their states but human readable logmessage speed up the process of accessing the state of a single target or test multiple targets in a simple correlation.

2.2.5 Power Profiling

For most embedded systems power consumption is of utmost importance - hence a state of the art testbed should provide means for power profiling a system. As the state of a target node can change quite fast, power profiling needs according to the Nyquist-Shannon sampling theorem to be at least double as fast as the states to be measured. This results in a sampling interval of 500μs which corresponds to a sampling rate of 2000 samples per second for this service (see Section 2.2.2). The power profiling is done entirely on the observer, no instrumentation is needed on the target node.
The design challenge of this service is hence to handle all incoming data fast enough and fill it into the database (see Section 2.3.3.1). Most of the time, one will not be interested in having all target nodes of the testbed being power profiled during the whole experiment period but rather having a subset of targets being profiled for specific time periods.

2.2.6 Further Services

In a service oriented architecture, it is easy to enhance the testbed with new functionality by adding services. A useful service would be to get dumps of the target node thus allowing for a checkpointing service as presented in [25]. A service that can, for example, write to the target parameters of a program which is then executed on the target is another idea for a future service.

2.3 Data Storage

On the observer, a possibly huge amount of collected data needs to be temporarily stored before it is synchronized with the test server which processes the data from all observers. The storage of this data needs to be persistent against power and network failures, reliable and flexible for structural changes. For this purpose, the two solutions at hand are using files or a database system.
Even though database systems are based on an existing filesystem they provide a broader set of functionality than a filesystem does. A database can store different kinds of information in one or more files and build a meta level by linking correlated information, whereas files can store information only in a line-by-line manner. We will focus thus on database systems as they can meet our requirements better than a filesystem can. Furthermore, the system used for storing data on the existing server is already organized in a database which will simplify the synchronization between the observers and the server.

A database has to be extendable, simple, and reliable. In order to provide reliability, a database needs to be transactional. Transactional databases fulfill the ACID properties; each operation on the database has to be Atomic, Consistent, Isolated and Durable from the point of view of the database user. These properties

have to be applicable even if the database crashes due to malfunction, power failure and such.

As embedded systems are limited in computing capacity and memory size the main requirements for a database running on the observer are the memory used by the application and the throughput. These factors are the main criteria when evaluating different database systems.

2.3.1 Design Requirements

Serverless databases do not have a service which connects clients to the database system. Each client wanting to operate on the database starts its own routine and accesses the database directly, not through a service. This has the advantage that the database needs less system resources but it requires the database system to take care about read and write permissions. In a database with a server the system service would do this. As our application needs to be fast and only the observers' data gathering service and the central collecting server access the database on the observer, it is advantageous to use a serverless database system.

The database is not stored at one place but distributed over the server and all observers. Thus, the local databases have to be accessible through different interfaces and hardware architectures. This requires a flexible and easy to handle database system that runs on different operating systems and can be accessed with the most common programming languages such as C/C++, Java, Phyton and so forth.

2.3.2 Database Architecture

From an architectural point of view, the database is split into two parts - one on each observer and the other one on the server. From a practical point of view, the databases on the observers and the server are standalone systems as they run in physically different locations and are not connected to each other. Thus, all constraints between these two systems, such as foreign keys, have to be checked by the software accessing the databases and not by the databases themselves as the database on the server cannot check the constraints on the observer database and vice versa. The only difference between the databases on the observers and the server is that the server database holds additional information about the test setup whereas the observer databases do not need to store this information because the observers are not aware of each other.

2.3.3 Database Model

The database model is independent of the database system used for implementing it as this model defines only the structure of the tables and relations between the tables. The database model can be seen in Figure 2.8.
The database structure on both the observers and the server are similar. Both

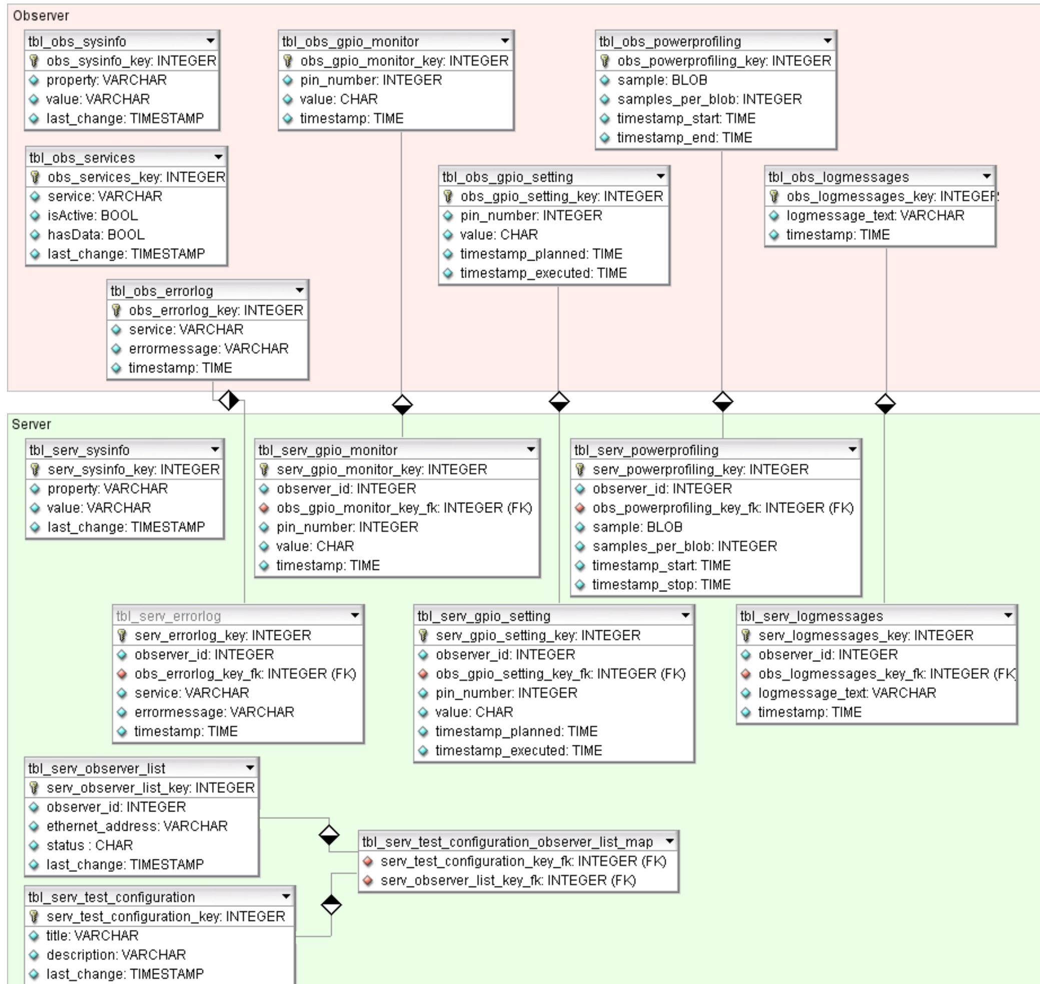


Figure 2.8: Database model, separated in observer (top) and server database (bottom). The relations between the top and bottom models are only virtual as the devices are physically and logically devided.

databases have tables with system information and one table for each testbed service. The server database has additional tables for the organization of test runs and attached observers. All tables have a primary key (*TablenameWithoutTblPrefix_key*) and a timestamp (*timestamp* or *last_change*). Tables on the observer carry the prefixes *tbl* and *obs* to their name, tables on the server carry *tbl* and *serv*.

2.3.3.1 Observer

The observer has a table for general system information (*tbl_obs_sysinfo*). This table holds an ID to identify the observer, information about the system version, and system location. Due to its general specification (fields *property* and *value*) the table can be used to store virtually any information needed for running tests.

The table *tbl_obs_services* holds two rows for each service - *isActive* to indicate that the service is active and *hasData* to indicate that data is possibly available in the table even though the service is not active anymore. This second flag is needed to indicate that there is still uncollected data in the table after the service has been shut down because the server collects data on the observers in intervals and thus cannot check directly after a service shutdown whether there is still data available. The flag *isActive* is set by the service when it starts up and reset when it stops. The flag *hasData* is set by the service after resetting the *isActive* flag. This approach with two flags is needed because it reduces the number of queries needed on the observer. Alternatively a trigger could be set on the service table to set the *hasData* flag after each insert operation but this would slow down insert operations significantly.
Each testbed service has its own table structure to store the service specific data. Except for the power profiling, each service stores one log value per row in its table. As the power profiling service runs with a high sampling rate and the data size per sample is low (16 Bit per sample), this service stores multiple samples per row. The service stores the measured samples in intervals (e.g. 1 second) as blobs[1] to the database, increasing the performance of the database significantly. To extract the samples from the blobs, the number of samples per blob (*samples_per_blob*) and the timestamps of the blobs' first and last sample (*timestamp_start* and *timestamp_end*) are stored. From this data, the timestamps of each sample can be recovered and it allows detection of missing samples.
Errors from a service, the kernel module or the database daemon are logged to the table *tbl_obs_errorlog*.

2.3.3.2 Server

The tables for the testbed services are identical to the ones on the observer except that the ID of the observer from where the data is originating is stored as a foreign key in each row to associate collected data with the respective observer.
The system information table is equivalent to the one on the observer. Furthermore there is a table with a list of all observers associated to the server (*tbl_serv_observer_list*) and their respective ethernet addresses. The field *status* opens up the possibility of using only a subset of the attached observers for a given testcase and to indicate broken observers.
In order to keep track of the used observers for a specific test, the table *tbl_test_configuration* lets the user define tests and table *tbl_serv_observer_configuration* opens the possibility of defining subsets of observers to be used in a specific test. To keep a history of the used observers in past test runs, a mapping table is defined to map the observer subsets to the testcases: *tbl_serv_test_configuration_observer_list_map*.

[1] A binary large object (blob) is a special data type used in databases to store data without formatting it.

2.3.3.3 Data Collection

The server periodically collects data from all attached observers. The collecting service is managed over a cron job running on the server. The data collection routine first accesses the table *tbl_serv_observer_list* and gets a list of all active observers. It then iterates through this list and gathers data from each observer.
To minimize the number of database operations on the observer, the data collector looks up the testbed services on the observer which are active or have data not yet collected (see Section 2.3.3.1). This is done by polling *tbl_obs_services*. The data collector then accesses the tables of all active services and gets the respective table key with the highest value. This value is then compared to the highest value stored on the server database. If there is a difference, new data is available on the observer. This data is fetched and inserted into the server database. Next, the data collector deletes all but one row on the observer (this is needed to ensure that during the next polling cycle, the data collector can get the value of the highest key) and resets the *hasData* flag if necessary.
The data collector is implemented in a way that allows to extend the set of tables which are synchronized.
If there is a need for additional tables to be synchronized this can easily be done by following these steps:

- Create the new table on the observer. It has to have set its name as *tbl_obs_tablename* and has to have a primary key defined as *obs_tablename_key*. Define triggers for timestamping if needed.
- Create the new table on the server. It has to have set its name as *tbl_serv_tablename*, has to have a primary key defined as *serv_tablename_key*, foreign key as *obs_tablename_key_fk*, and a field *observer_id*. Define triggers for timestamping if needed.
- Insert a row in *tbl_obs_services*. Set the *tbl_obs_tablename* as *service*.

2.3.3.4 Database Administration

The only database parts that need to be administrated by the user are the table *tbl_obs_sysinfo* on the observer and tables *tbl_serv_sysinfo*, *tbl_serv_observer_list*, *tbl_serv_test_configuration*, and their mapping-table *tbl_serv_test_configuration_observer_list_map* on the server.
On the server, a graphical user interface (GUI) should be used for managing the tables as changes in the observer lists are not trivial. The GUI is preferably a webinterface or Java application.

Chapter 3

Implementation

3.1 Data Storage

3.1.1 Database Software

Large database applications such as MS SQL, Oracle or MySQL take up large amounts of memory and processing capacity and are not serverless either.
A few databases are suitable for embedded systems, SQLite and H2 being among them. Both databases are transactional, serverless, and have a small footprint. Both databases are widely used in commercial and academic applications.
For the testbed and especially the observer, two factors are important regarding the implementation of a database: throughput and memory usage on the filesystem. These two factors are the ones that are most limited on an embedded system. For a database, throughput is defined as the speed at which data can be inserted into the database. This includes opening the database tables, checking constraints, possible trigger actions, inserting the data, updating database tables, closing the database. On both, SQLite and H2, throughput and memory performance tests have been carried out.

3.1.1.1 SQLite Database

SQLite is an embedded SQL database engine that is serverless and fully ACID-compliant (see Section 2.3). All data is stored in a single file that is portable to all platforms due to its architecture-independent design.
A specialty is the manifest typing implemented in SQLite. The vast majority of database systems use static typing: a data type is assigned to each column in a table and every operation on that column checks whether the data is of the correct type. This has the advantage that the reader of a column always knows what kind of data is stored in it. With very few exceptions[1], SQLite uses manifest typing: a column can be declared to have a certain data type but this is not checked on

[1] One exception are columns declared as INTEGER PRIMARY KEY which always have to hold integer values.

operations on this column. This brings the advantage that each value of a column only takes as much memory space as it needs for storing the data and not as much as the data type defines (e.g. a value consisting of one character stored in a column declared as VARCHAR(100) takes only one byte instead of 100 bytes in a static typing architecture).

SQLite can be used from virtually any program that is able to access a filesystem. Various SQLite APIs exist for different programming languages.
SQLite is a library that uses very little space - making it an ideal choice for embedded systems such as the WSN testbed. The library has a memory footstep of less than 380KB which can be tuned to less than 180KB if unused functionality is omitted.
Due to its very compact size, robustness, and efficiency, SQLite is widely used in a variety of commercial and academic applications. Examples are Android, Symbian, Mac OS-X and Firefox.

3.1.1.2 H2 Database

H2 is lately being used in a variety of projects such as the Global Sensor Networks project GSN.
H2 is a Java based application and can thus not be addressed easily from within languages other than Java.
H2 is lightweight - its libraries need less than 1MB of memory, it is fast and opensource. One of the advantages of H2 compared to SQLite is that it can be used serverless as well as with a server process. This opens the possiblity of letting the databases on the observers run in embedded (serverless) mode while the data gathering server operates in server mode to provide easy access for applications and users.
H2 implements security principles such as password authentication, file encryption, and SSL/TLS. This feature is currently not needed on the WSN testbed but could be useful in the future.

3.1.1.3 Evaluation

Both databases have been tested for the throughput of insert operations. Insert operations are important because for the testbed it is the most used database operation: the observer continuously inserts data to its database, the server gets data from all observers and inserts it into the server database.
A defined number of rows, each containing three integer values, were inserted as one single transaction in the databases in three consecutive runs. A test consisted of three runs to allow speed improvements induced by Java caching. Each test was run on a standard PC[1] and the Gumstix platform. The databases were stored in flash memory as this is the only accessible memory (besides RAM) on the Gumstix

[1] Quad core PC (4x2.4GHz, 3.24 GB RAM, Windows XP)

Platform	PC			Gumstix		
Database	SQLite		H2	SQLite		H2
Access method	Bash	Java	Java	Bash	Java	Java
1000	296	24	81	1110	53760	6192
2000	212	51	128	2063	108564	10774
25000	871	473	474	28413	1424423	165371

Table 3.1: Runtime of n insert statements on the database in [ms].

Platform	Gumstix	
Database	SQLite	H2
1000	1242	190
1250	1750	168
2500	1216	129
5000	883	112

Table 3.2: Runtime for insertion of 5000 samples split in blobs with n samples per blob in [ms] using Java.

platform. Access was made through Java (H2), the Java wrapper Zentus (SQLite) and Bash scripts (SQLite). For SQLite, the no-sync mode[1] was used as this provides higher speed. H2 ran in the embedded mode[2].
As can be seen in Table 3.1, execution time increases approximately linear with the number of inserted rows when measured on the Gumstix platform. Measured on the PC, the execution time is sublinear. This effect is most likely due to the fact that on the PC with its fast CPU, the execution time is dominated by the administrative overhead of the database system (e.g. opening the database or gaining and releasing access rights) rather than the actual insert operations whereas on the Gumstix the insert operations and thus the file system writes dominate the execution time.

In a second test, blobs were used instead of integer values. For our application this allows to store the incoming stream of binary measurement data from the powerprofiling service in an efficient way.
The powerprofiling service generates a maximum of 5000 samples per second which are inserted as blobs into the database. For the throughput performance test, blobs with a specific number of float values were continuously inserted in each database. The average over a ten second period can be seen in Table 3.2. The SQLite test was run using Java as the access method.
From these data, the time needed per sample can be calculated according to Table 3.3. As can be seen, H2 is about ten times faster than SQLite when the databases

[1]SQLite can be configured to stop operation at critical moments to check whether the processed data has actually been written to the disk. In no-sync mode, SQLite never makes this check and thus provides higher throughput at the cost of reduced robustness.
[2]If H2 runs in the embedded mode, the database is serverless and thus faster.

Platform	Gumstix	
Database	SQLite	H2
1000	0.248	0.038
1250	0.350	0.034
2500	0.243	0.026
5000	0.177	0.022

Table 3.3: Time to insert one sample calculated from Table 3.2 in [ms]. As access method, Java is used.

	Raw data size	SQLite		H2	
		DB size	Overhead	DB size	Overhead
3000 integers	11.72	14	19.45	416	3449.49
6000 integers	23.44	26	10.92	512	2084.30
75000 integers	292.97	311	6.15	3501	1095.00
Blob with 5000 floats	9.77	52	432.24	56	473.18

Table 3.4: Data size in [KB], overhead in [%].

are accessed using Java.

The memory requirements have been tested for both databases. Data with specific size was inserted into the databases[1] and the size of all database files was determined. The overhead was calculated as the ratio between the size of the inserted data and the size of the database. The smaller the overhead, the more efficient the database in terms of memory usage.
As can be seen in Table 3.4 the tested database systems behave very differently in terms of memory usage. While they both take approximately the same amount of memory for storing blobs, H2 is less efficient when storing relatively few integer values. Both databases scale approximately linear with H2 having a big initial overhead.
SQLite stores all data in a single file while H2 creates three files: one each for data, index and log file. When storing blobs, H2 creates even more files since it outsources the blob data to external files.

3.1.1.4 Comparison and Software Decision

The database being used for our application should be small, fast, reliable, extendable, simple, and versatile. Reliability, simplicity, and extendability are met by both databases, with SQLite being more versatile because it does not depend on Java as the only access method.
According to Table 3.4, SQLite manages to store data with much less overhead

[1] The access methods are of no concern as the generated data is the same no matter which language or wrapper is used.

than H2 making it more efficient in terms of memory usage.
H2 has a higher throughput compared to SQLite with the Java wrapper which makes it the preferred database when using Java only. As the services running on the observer will be written in C, Bash, Java and possibly other languages (e.g. Perl) SQLite is preferred over H2 as it is more versatile and at least as fast as H2 as long as it is not used with Java. For SQLite, the memory footprint of the library as well as the memory efficiency to store data in the database are much lower.
Moreover, SQLite provides functionality to easily access the database from a shell. This can be useful for quick inspection of gathered data or debugging purposes.

3.1.2 Server Implementation

The server is implemented using SQLite as the database system and a standard PC running on Windows[1]. The database is set up according to the database model defined in Section 2.3.3. As SQLite does not provide automatic timestamping of rows, triggers are defined on all tables which require automatic timestamps.
Two cron jobs are defined: one that periodically cleans up the database to free unused memory and keep the footprint of the database as small as possible. The other cron job is the data collector which periodically collects data from all observers.
Both cron jobs as well as the initial setup of the database are configured and administrated with the help of a bash script which was written for that purpose.

The data gathering script is invoked in a configurable interval. Its purpose is to connect to all observers that are marked to participate in a certain test setup and to gather all data pending for download on the observers. The algorithm works as follows:

1. Select all active observers from table *tbl_serv_observer_list* on the server database.
2. If no active observers: stop.
   Otherwise: get next active observer.
3. Select all tables on the observer that have data to fetch.
4. If no table: repeat 2.
   Otherwise: get next table.
5. Select maximum $key_{observer}$ of current table on observer.
6. Select maximum key_{server} of corresponding table on server.
7. Compare keys:
    - $key_{observer} \leq key_{server} \Rightarrow$ repeat 4

[1] Cygwin is installed in order to use Unix programs such as SSH, Cron or Bash scripts.

- $key_{observer} > key_{server} \Rightarrow$ goto 8

8. Get all rows from the current table and store them in the corresponding table on the server.

9. Delete all rows of the current table on the observer up to but excluding row $key_{observer}$.

10. Repeat 4.

This algorithm minimizes the number of accesses to the observer as only tables are queried which actually have data to report. In step 9, keeping the row with the maximum key on the observers' database ensures that during the next synchronisation of the table, the maximum key can be fetched in step 5.

3.1.3 Observer Implementation

The database on every observer is implemented using SQLite. The database is stored on an SD flash card as the database size can grow large. The structure of the database is defined according to the database model from Section 2.3.3. As SQLite does not provide automatic timestamping of rows, triggers are defined on all tables which require automatic timestamps.
As on the server, a cron job is defined when initially setting up the database which runs periodically and cleans up the database to free unused space. This is especially important on the observer as the database size can become quite large when all services are active and reporting to the database. After all data is fetched from the server, the database size shrinks to almost zero before it blows up again. It is thus good practice to clean up the database periodically to free unused memory.

3.2 GPIO Setting Service

The goal of the GPIO setting service is to be able to control the behavior of the target node by issuing commands over wired GPIO pins of the microprocessor. One GPIO line can be set or cleared by the observer which triggers some user defined action on the target node.

3.2.1 Overview

With the GPIO setting service, the user can schedule an event to set or clear a pin at a defined time in the future. The service keeps track of all scheduled events, executes them and reports the actual execution time back to the database. If an event cannot be executed for whatever reason, this is logged as well.
Because the setting of the pin has to be as accurate as possible, it is implemented in a kernel module to provide best possible time accuracy. A pin setting or clearing

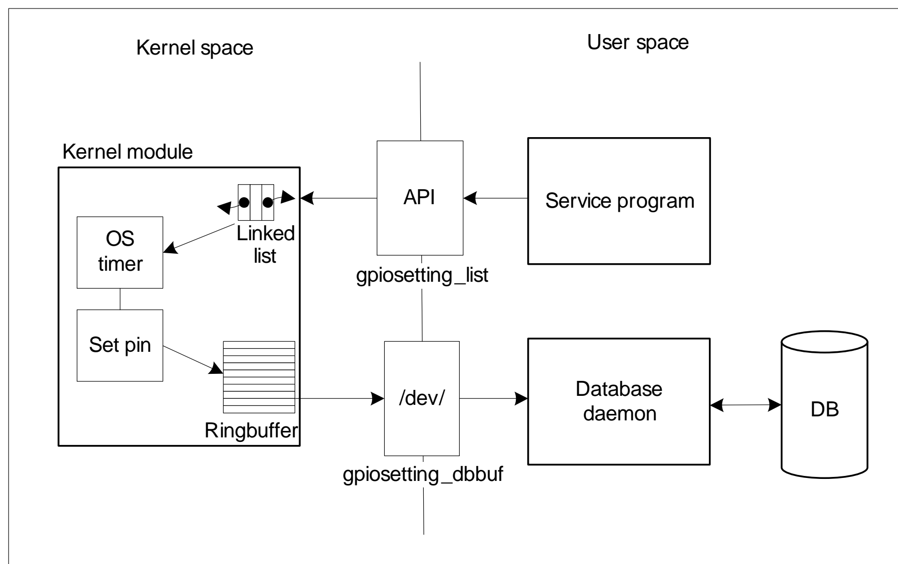


Figure 3.1: Overview of components of the GPIO setting service.

event is scheduled with the help of an OS timer which creates an interrupt after the specified time. The corresponding interrupt handler sets or clears the pin, takes a timestamp and reports to the database.
An overview of the components of the service is illustrated in Figure 3.1. The service has three components: a kernel module for the functions with timing constraints, a user space program for implementing the API and a database daemon in user space which inserts the processed events into the database. All time critical functions are implemented in the kernel module.

3.2.2 API

The API of the service follows the general service layout (see Section 2.2). There are five API commands:

- *testbed_gpiosetting_start*: Start the service by loading the kernel module and starting the database daemon.

- *testbed_gpiosetting_stop*: Stop the service, perform cleanup functions, stop the database daemon and unloading the kernel module.

- *testbed_gpiosetting_add(pin, edge, time)*: Schedule a new event. This API command requires three parameters:

  - *pin*: Pin number of GPIO on Gumstix.
  - *edge*: 0 to clear the pin, 1 to set it.

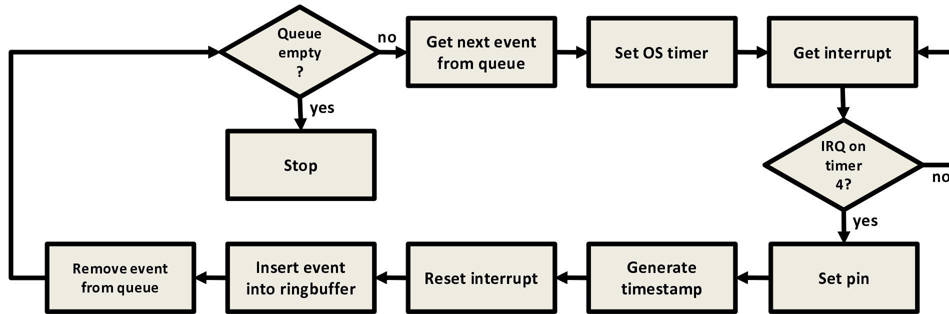


Figure 3.2: Program flow of kernel module after inserting a pin setting event. After removing an event from the queue the kernel module schedules the next event if there is at least one in the queue. Error checking is not showed in the flow chart.

  – *time*: Absolute timevalue at which the event shall occur. Relative timevalues are not accepted and have to be converted to absolute values by the user.

- *testbed_gpiosetting_remove(pin, edge, time)*: Remove a scheduled event from the queue. The three parameters to be provided are the same as for the add-command.

- *testbed_gpiosetting_list*: List the event queue in sorted order.

3.2.3 Kernel Module

The kernel module does the most important work for the service. At startup, it registers two devices in `/dev/`: `gpiosetting_list` and `gpiosetting_dbbuf`. Both devices are needed to transfer data from user to kernel space and vice versa. The list device `gpiosetting_list` transfers IOCTL commands from the user API to the kernel and lists the event queue from the kernel if requested by the user space API *testbed_gpiosetting_list*. The dbbuf device `gpiosetting_dbbuf` is accessed by the database daemon. The daemon reads the ringbuffer with all processed events from the kernel module using this device and reports them to the database. When the service stops, both devices are unregistered.
In Figure 3.2, the program flow of the kernel module after inserting a schedule for a pin setting event is shown.
The module is in an idle state until there is at least one event registered. All events to process are stored in a linked list, sorted by execution time. If the list is not empty, the kernel module gets the first entry and calculates how much time is left between the current time and the execution time. Next, the operating system timer channel 4 of the Gumstix' PXA270 processor is activated and set. An OS timer is a set of registers and corresponding functionalities to provide high precision time measurement on the hardware level. According to [26] the steps to set up an OS timer are the following (see also Figure 3.3):

Figure 3.3: Steps to set up an OS timer of the PXA270 processor in the Gumstix. The boxes represent the PXA270 registers that need to be written. An x denotes a channel-specific register.

1. Enable interrupts for the desired channel by setting the corresponding bit in the OS Timer Interrupt Enable Register (OIER). If set, an interrupt is generated if a match occurs between the OSMRx register and the OS timer.

2. Set the match value using the OS Timer Match Register (OSMRx) of the channel. This is the value that is compared against the count register OSCRx. The OSMRx register holds an absolute time value.

3. Set up the timer to compare against. The OS Match Control Register (OMCRx) lets the developer define against which count register to compare, what actions should be taken after a match and which counter resolution to use.

4. The register of the OS Timer Count Register (OSCRx) increments on rising edges of the clock which was selected to be used in the previous step.

5. If a match occurs (meaning that the OSCRx has the same value as the OSMRx register), the corresponding bit for the channel where the match occured is set in the OS Timer Status Register (OSSR). The interrupt handler has to reset this bit in its routine.

The interrupt handler first checks whether the interrupt occured for channel 4. If it did, the GPIO, which has to be set or cleared, is first put into output-mode and then set to the appropriate value. Next, a timestamp is taken to record the execution time of the GPIO setting event. This information is stored in a struct that holds all event data. Next, the interrupt register of the OS timer is reset and the struct is removed from the event queue and inserted into the ringbuffer which holds all processed events not yet fetched by the database daemon.
The insertion process has to check whether a wrap around in the ringbuffer occured and report this to the database daemon, if appropriate. A wrap arround occurs whenever the kernel module is much faster at inserting processed events compared to the database daemon reporting the events to the database. Both processes update a pointer to the next position they are going to operate on. If the kernel module pointer overtakes the database daemon pointer, a wrap around occurs inducing a dataloss because information that was not yet reported to the database is overwritten. In this case, an error message reporting the wrap around is generated and reported to the database.
After inserting the event into the processed events buffer, the interrupt handler can

safely delete the event from the event queue and schedule the next event.

The event scheduler always gets the first event from the event queue. It then checks, whether the event was already missed for some reason. If yes, the execution time field of the struct that holds the event is filled with zeroes to indicate a missed event. The event is then removed from the event queue and inserted into the ringbuffer holding the processed events before scheduling the next event.
If the event has not already been missed, the value for the OS timer match register is calculated and the OS timer is activated. If the queue is empty, the OS timer is deactivated and the kernel module returns into its idle state.

3.2.4 Database Daemon

The database daemon runs in user space in the background. Its purpose is to get the list of processed events and possible error messages from the kernel module and report this data to the corresponding tables in the database[1].
Therefore the daemon gets the list of processed events from the kernel modules' ringbuffer over the device `gpiosetting_dbbuf`. The device is read in blocking mode, so the kernel module puts the daemon to sleep as long as there is no data available on the device. As soon as there is data, the kernel module writes it to the device and wakes up the daemon which reads the data. The daemon itself keeps an adjustable buffer which allows for periodic and thus more efficient writing to the database. As soon as the buffer is full or after a timeout, the daemon starts a transaction and writes all data from the buffer to the database. Using transactions, the daemon can minimize the database accesses as well as the transaction time because transactions are more efficient to perform than single insert-operations to the database.

When the service is shutting down, the daemon catches its termination signal, flushes the buffer to the database and unregisters the device file.

[1] Processed events are reported to the table *tbl_obs_gpio_setting*, errors of all services are logged in table *tbl_obs_errorlog*.

Chapter 4

Service Evaluation

Software implementations need to be evaluated in order to show that they meet their requirements. In this master thesis, the GPIO setting service (see Section 3.2) has been implemented, tested, and finally evaluated according to its most important requirement: the accuracy of the pin setting mechanism. It is crucial for this service to set a pin as precisely as possible compared to the planned time of the setting event. Two values determine the accuracy of the service:

- Offset. The offset is the amount of time the service is late or early when setting a pin compared to the time it was planned. The constant offset can be internally compensated.
- Variance. The variance is the variation of the offset which is induced by unpredictable factors. The variance cannot be compensated as it is not known before the execution time of an event.

Offset and variance of the GPIO setting service are influenced by the internal offset of the observer which performs the pin setting action. Whenever the service sets an OS timer on the observer, no guarantee can be given regarding the timer timeout - the operating system only guarantees that the interrupt is not triggered before the planned time. This is mainly because there are possibly other interrupts with higher priority to be handled by the CPU before handling the OS timer interrupt. Moreover, once the OS timer interrupt is handled, the pin cannot immediately be set as there is a certain amount of overhead in the interrupt handler.
The clock of the observer does not tick precisely but has a certain skew due to inaccuracies of the oscillator that drives the clock. This skew is in the range of $\pm$50ppm for the Gumstix [22]. If the mean skew is positive or negative, the clock will drift, meaning that it is either slower or faster compared to real time. This effect can partly be compensated by using a time synchronization protocol to minimize the drift. For this purpose chrony was used and, consequently, had to be evaluated in order to determine its influence on the internal accuracy of the observers as well as on the accuracy between the observers, as they have to have a common timebase in order to deliver inter-observer-comparable measurement data.

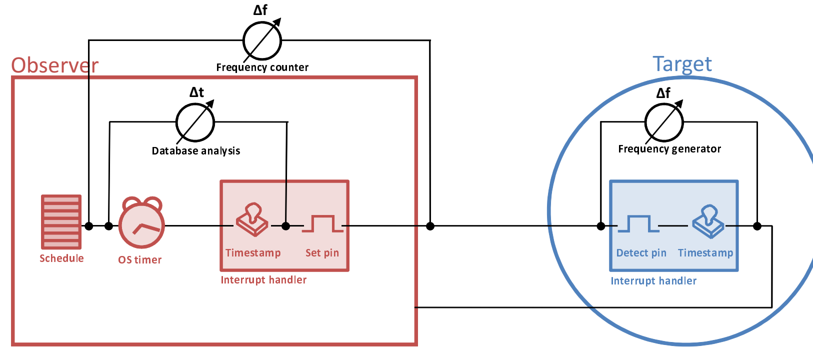


Figure 4.1: Instrumentation of the testcase. Offset measurements are labeled with Δt, variance measurements with Δf.

Several measurements were conducted on both the observer and the target node to determine value and impact of offset and variance to the accuracy of the service.
As the observer as well as the target node are embedded systems, measurements can only take place at certain physical points of the system which are suitable for instrumenting. The measurements that were performed can be seen in Figure 4.1.

The offset was measured on the observer by evaluating the contents of the observers database which allowed for calculation of the offset between the time an event was planned and the time it was actually executed.

The variance was measured on the observer and the target node. As it cannot be directly obtained, it was measured using a frequency generator[1] and frequency counter[2] respectively. On the target node, the frequency generator was used to produce periodic signal changes which can be interpreted as periodic pin changes. The target node timestamped the detected pin changes. From these timestamps a frequency could be reconstructed and the error of the frequency between the generator and the target node could be determined. On the observer, a series of periodic pin changes was scheduled and measured with the frequency counter. From the result of the frequency counter measurement and the known interval between pin changes, the frequency error could be calculated.
A measured positive or negative mean frequency error means that the clock of the device under test is not running accurately compared to real time and therefore a drift is resulting. This can be better explained as follows: a low-frequency signal running in the same space as a high-frequency signal leads to a drift between the edges of both signals as can be seen in Figure 4.2. As the offset between the signals grows over time, the result shown in Table 4.1 imposes a drift in the target's response time to a pin setting.

[1]HP 33120A Function Generator / Arbitrary Waveform Generator
[2]Agilent 53131A Universal Frequency Counter

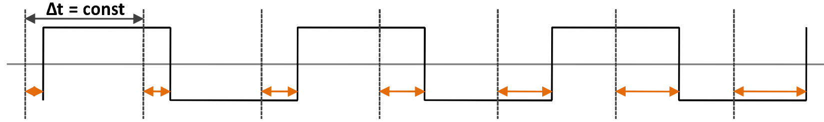


Figure 4.2: Drift of signals with different frequencies. The offset between the edges of the signals grows over time.

Mean error [mHz]	-0.010
Standard deviation [mHz]	0.017

Table 4.1: Mean error of measured frequency when detecting pin changes on target node induced by the function generator at 0.5Hz. The mean error is the average error of the measured frequency compared to the induced frequency.

All measurements on the observer were made under no load of the CPU as well as under full load as this can cause side effects that need to be addressed. The observer CPU was loaded by running an SSH key generation with a large key size which brings the CPU to full load but does not produce any interrupts.

The variance measurement on the target and on the observer are discussed in Section 4.1. The evaluation of the offset on the observer is handled in Section 4.2.

4.1 Pin Setting Variance

The variance of the pin setting was measured at two positions: on the observer when setting pins and on the target node when detecting pin changes.

4.1.1 Target Node Variance

For measuring the variance of the target node, periodic events at 0.5Hz were generated using the frequency generator. The generated frequency was applied to GPIO pin 13 of a TinyNode using breakout wires and a BNC cable. The TinyNode was set up to sense state changes on pin 13, timestamp the events and report them to the observer over the UART interface using the Java SerialForwarder. From the timestamps the frequency of the pin changes could then be calculated and compared to the frequency set with the generator. The results of the target node measurements can be seen in Table 4.1. A small negative error of the frequency is measured. This means the frequency is lower and hence the period is longer compared to the induced frequency. Thus, the target nodes' clock is running slower than it is supposed to.
The standard deviation of the mean error is low which leads to the conclusion that

	No Load		Full Load	
	Mean error [mHz]	Standard deviation [mHz]	Mean error [mHz]	Standard deviation [mHz]
C-Program	-0.013	0.003	-7.326	9.054
Kernel Module	-0.013	0.000	-0.009	0.001

Table 4.2: Mean error of generated frequency on observer at 0.5Hz measured with the frequency counter. The userspace C-program runs with the default scheduling priority. The kernel module runs with time synchronization turned off to avoid disturbances due to clock adjustments.

	No Load		Full Load	
	Mean error [mHz]	Standard deviation [mHz]	Mean error [mHz]	Standard deviation [mHz]
C-Program	-5.496	122	-45076	284
Kernel Module	-1.500	3.635	-1.501	4.681

Table 4.3: Mean error of generated frequency on observer at 50Hz measured with the frequency counter. The userspace C-program runs with the default scheduling priority. The kernel module runs with time synchronization turned off to avoid disturbances due to clock adjustments.

the mean error is almost constant.
In addition to the drift there is an offset in the response time induced by the time needed to invoke the interrupt handler and to timestamp the event in TinyOS. According to [27] a context switch to an interrupt handler needs 20 cycles in TinyOS which corresponds to 2.5μs on the TinyNodes' MCU. This value represents a lower bound for the offset which will be larger the more load there is on the MCU.

4.1.2 Observer Variance

The variance of the pin setting on the observer was measured using the frequency counter and a sequence of pin setting events at different frequencies (0.5Hz and 50Hz). GPIO pin 75 was used in the observer's CPU for this purpose and was connected to the frequency counter using breakout wires and a BNC cable. The frequency counter was set for DC coupling with a 100kHz filter and triggering at 1.5V on the positive slope.
The results of the variance tests on the observer for different frequencies can be seen in Tables 4.2 and 4.3. With the CPU under no load and a low frequency of 0.5Hz, the mean error is for the kernel module as well as for the userspace C-program small with a small standard deviation.
For tests under full load of the CPU at 0.5Hz, one can see the limits in the throughput of the userspace C-program: the CPU is busy all the time and as the C-program has the same scheduling priority as the SSH keygeneration program used for putting the CPU to full load, the C-program suffers from large delays as it does not always

get the CPU when it needs it. This effect can also be observed when measuring at higher frequencies (see Table 4.3) where the userspace module becomes unusable.
These test results suggest using a kernel module for the pin setting service, as code in the kernel space runs in interrupt context and can thus preempt userspace code. Another method to overcome this problem could be to increase the scheduling priority of the userspace C-program but one would risk starving other processes running on the CPU by assigning a high priority to the program. This is not likely to happen with a kernel module as only the time critical part of the module (the pin setting) runs in interrupt context whereas the rest of the kernel module is preemptable. Nevertheless the kernel module can also starve other processes if there is a high number of pin setting interrupts to handle.
The standard deviation of the kernel module is good even with high frequencies and under full load of the CPU as can be seen in Table 4.3. The mean error of -1.501mHz corresponds to an error in the time domain of about 0.6μs which is very small.

4.2 Pin Setting Offset

In Section 4.1 only the drift and standard deviation of the pin setting signal were tested and discussed. These measurements do not allow for analysis of the offset induced by the processor to switch to the interrupt handler which sets the pin and actually does the setting. This offset is inevitable and thus cannot be reduced. However, as the offset is known before execution time it can be eliminated in the software by subtracting it when scheduling events for the OS timer.

4.2.1 Test Setup

The test is done using software measurements. A number of pin setting events are scheduled with different intervals between the events on a CPU under no load as well as on a CPU under full load. To ease comparisons between the different tests, the same frequencies as in the other tests (0.5Hz, 50Hz, 100Hz) were used. These frequencies correspond to intervals of 1s, 10ms, and 5ms between the events.
The interrupt handler did not need any modifications as directly after the pin setting a timestamp is taken which is reported to the database on the observer. This makes it easy to analyse the offset of the pin setting as one can compare the planned time to the execution time which are both stored in table *tbl_obs_gpio_setting* in the database (see Section 2.3.3).

4.2.2 Test Results

The testresults for the measured frequencies and the different CPU loads can be seen in Figure 4.3. The offset is always positive as the OS timer guarantees to trigger the interrupt at earliest at the planned time.
The offset itself is explained by the time the processor needs to possibly wait for

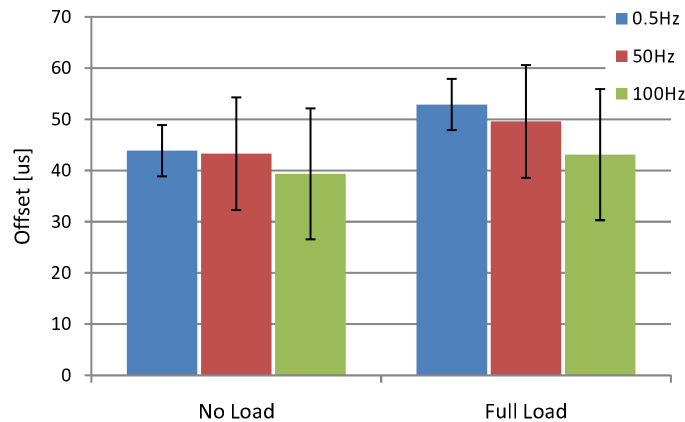


Figure 4.3: Offset of pin setting on observer. The results are derived from averaging the offset time of 6000 pin changes. No time synchronization was active during the measurement.

other interrupts to finish, to perform the context switch to the interrupt handler (6 cycles≈33ns according to [26]), and to execute the interrupt handler code to the point where the pin is set and afterwards the timestamp is taken (approximately 14μs). As the time needed for the context switch is negligible, the deterministic part of the offset can be determined as approximately 14μs.

The source for the remaining part of the offset time could not be determined and can have several causes. One is the fact that the interrupt handler of the OS timer has to wait if another interrupt with a higher priority is running its interrupt handler at the time the OS timer interrupt occurs. This can also be a possible explanation for the increasing standard deviation for high frequencies: the higher the frequency (and hence the shorter the interval between two occurrencies of the OS timer interrupt) the higher the probability that the OS timer interrupt triggers when another interrupt handler is already running.

Another factor that can prolong the offset time is the imprecision of the 13MHz clock on the observer. The 13MHz clock has a specified frequency tolerance of ±50ppm [22]. The ARM core that was used for the testing had a frequency error of -26.7ppm (±0.3ppm). The frequency error of the observer was determined by examining the logfiles of the time synchronization tool chrony which measures the frequency error of the clock. An error of -26.7ppm corresponds to an error of 26.7μs per second. This means that even though the frequency error lies in the limits of the specification, the clock which is used to schedule the pin setting events is running slow compared to real time. This leads to an error in the timestamping mechanism of the service that becomes larger the farther in the future a scheduled event lies (the frequency error sums up over time). This effect explains the decreasing offset values for increasing event frequencies in Figure 4.3.

4.3 Time Synchronization Effects

Time synchronization tools compensate for the clock drift induced by the frequency error of the CPU clock. This is done by slowing down or speeding up the logical clock.
When running tests for the pin setting accuracy and pin setting processing time evaluations, timelines of several measurements showed peaks, as can be seen in Figure 4.4(a) or level changes, as in Figure 4.4(c) about three seconds after the start of the measurement. When turning off the time synchronization tool chrony which runs on the observer, these level changes disappeared. It is thus evident that chrony has an influence on the execution time of events.
As chrony is based on NTP it adjusts the clocks in periodic intervals of four seconds by switching to one of two possible correction frequencies of the local clock. This can be also seen in the logfiles of chrony which are updated every 64 seconds. Every 60 seconds there is a synchronization of chrony with the time server followed by a four second adjustment interval. The correction frequencies are achieved by swallowing a specific number of oscillator pulses in the clock prescaler [23].
In Figure 4.4(a) the peak is high for about four seconds followed by a mean error that is almost zero. This is most likely an adjustment interval of chrony that was recorded and influenced the measurement. As after the adjustment interval the clock is running accurate, the mean error drops to almost zero. The level change in Figure 4.4(c) lasts for more than four seconds. This could be explained by a big offset of chrony where after the first adjustment interval another adjustment interval needed to be performed in order to correct the clock drift.

Even though the time synchronization mechanism accounts partly for inconsistent timestamps on the observer, it is an integral part of the testbed as it is needed to synchronize all observers which have to have a common timebase in order to deliver inter-observer-comparable timestamps. In the next section, the timing behavior of several observers against each other is analyzed.

4.4 Synchronicity of Observers

All evaluations in the previous sections were carried out on one single observer-target-subsystem and could therefore only capture the different internal errors of the observer, the target node and their interconnection. As in the testbed numerous observers will operate together, they need to have a common timebase which allows for synchronous execution of commands such as GPIO pin settings. Furthermore, the timestamps generated on different observers should be comparable and thus require synchronized clocks on the observers as well.

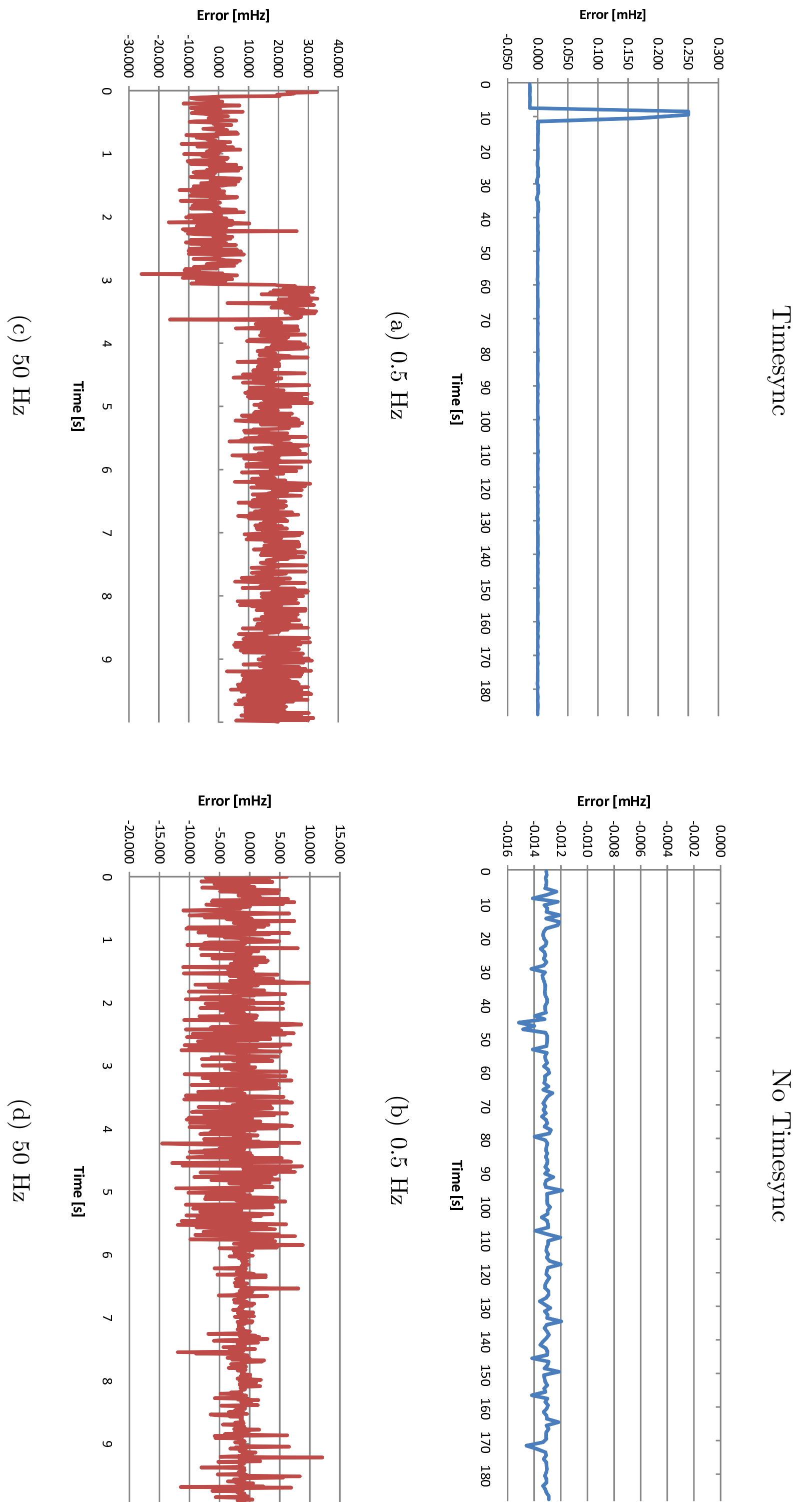


Figure 4.4: Influence of time synchronization on evaluation experiments. The charts represent typical measurement results for pin setting accuracy testruns with and without the time synchronization tool chrony running.

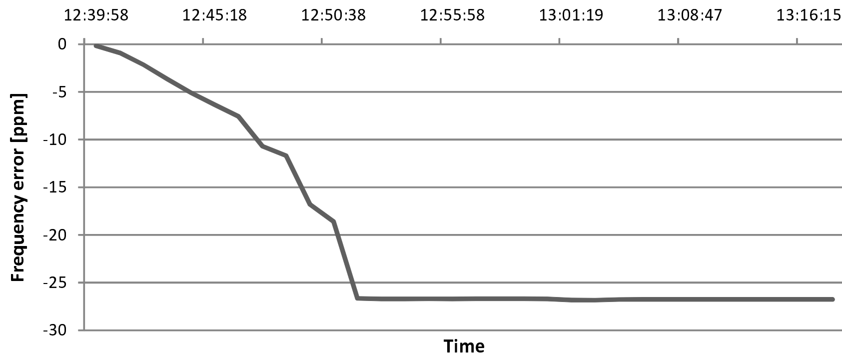


Figure 4.5: Time synchronization of chrony at system startup. Chrony connects to the server once minute and measures its offset to the time server. From the offset, the clock drift (frequency error) can be calculated and the clock can be adjusted accordingly. As can be seen, chrony needs around 14 minutes to stabilize.

4.4.1 Test Setup

Two observers with identical configuration have been compared against each other. Eventhough there will be a potentially high number of observers operating at the same time in the testbed, it is sufficient to evaluate the synchronicity of two observers as all observers on the testbed will be synchronized over the same server and not amongst each other. This implies that the observers are independent of each other and thus the average error between two observers is constant no matter how many observers are operating in the testbed.
As timing server, a standard PC was used which synchronized itself using three time synchronization servers. This server was then used as a local time synchronization server for the attached observers. In order to minimize the network latency, the server and all observers were attached to the same network switch. Before each measurement, the observers were running idle for at least 30 minutes to let their time synchronization settle down to steady state.

Two different measurements were carried out: one to measure the time offset when multiple observers are scheduled to set a GPIO pin at the same time and another to measure the time offset when monitoring a GPIO pin at multiple observers.
For measuring the time offset when multiple observers are scheduled to set a GPIO pin at the same time, the two Gumstix computers at hand were connected to an oscilloscope[1]. Both observers were scheduled to execute 1000 pin setting events according to the same schedule. The trace of the pins was then recorded with the oscilloscope and the offset between the edges of the two signals was calculated from this data.
In the second set of measurements, the offset of the timestamps when monitoring

[1] Tektronix MSO 4054 Mixed Signal Oscilloscope

	No Load	Full Load
Mean offset [μs]	12	17
Standard deviation [μs]	9	4

Table 4.4: Time difference between two observers when setting pins at synchronized time intervals.

	No Load	Full Load
Mean offset [μs]	14	156
Standard deviation [μs]	47	38

Table 4.5: Time difference between two observers when monitoring pins.

the same pin was measured. For this test, the frequency generator was connected to two observers. Using the generator, a sequence of 1000 pin changes was triggered on both observers at the same time. The observers were set up to detect the changes of the pins using the gpio-event kernel module [28] and output the generated timestamps. Analyzing these timestamps allowed for comparing the offset of the time synchronization between the observers when detecting events.
Both test scenarios were conducted under full load as well as with an idle CPU on both observers.

4.4.2 Test Results

The results of the testruns with the oscilloscope can be seen in Table 4.4. The mean offset between the observers is of about the same magnitude for both an idle CPU and under full load. This makes sense, as under full load both observers are busy and will serve the pin change interrupt at about the same time, as this measurement does not take the internal offset of the observers into account due to their CPU load.

The results of the testruns for pin monitoring synchronicity are listed in Table 4.5. There is a difference in the offset for an idle CPU compared to full load of a factor 10. This can be explained by the mechanism that was used for measuring the offset: each pin change triggers an interrupt in whose interrupt handler the timestamp is generated. As according to [26] interrupts of GPIO lines have low priority, there is a high chance that under full load the CPU is handling other interrupts with higher priority when a GPIO interrupt arrives. This effect cannot be seen in the results for the GPIO setting tests (see Table 4.4) because there the pin is set in the interrupt handler of an OS timer which has a very high priority.

4.5 Conclusions

The most important factor when evaluating the GPIO setting service is the accuracy of the pin setting which dictates an acceptable error for the service of at most

	Mean error [mHz]	Standard deviation [mHz]
Observer	-1.501	4.681
Target	-0.010	0.017
Total	-1.511	4.681

Table 4.6: Worst case variance of GPIO setting service. The total standard deviation is calculated as $\sigma = \sqrt{4.681^2 + 0.017^2}$ as both variables are independent and normally distributed.

500μs.
The error of the service can be distinguished into two factors: the offset and the variance of the error. The offset can be partly compensated by the service software whereas the variance is stochastic and thus cannot be compensated.
The maximum offset measured on the system was 53μs (see Figure 4.3). This is far below the requirements and thus of no concern.
The variance of the pin setting on both the target node and the observer are in the range between microhertz and a few millihertz. Summing up the worst case variances of the observer and the target (see Sections 4.1.2 and 4.1.1), according to Table 4.6 one gets a worst case variance of -1.511mHz with a standard deviation of 4.681mHz. The variance corresponds to an error in the period of about 0.6μs (if calculated for a measurement frequency of 50Hz). This value as well as the offset is far below the requirement.
Summing up, it can be stated that the evaluated observer fulfills the requirements as the variance as well as the offset are far below the requirement.

The evaluation of the time synchronization between two observers shows that the offset between two arbitrary observers is in the range of 150μs or less (see Section 4.4.2). As the observers will not be under full load all the time, generally there will be a clock difference of less than 20μs between them. This time synchronization precision is within the limits accepted for the testbed. Further precision could be achieved if necessary by tuning the real time clock (RTC) on the observers.

Chapter 5

Conclusion and Future Work

All software implemented in this thesis - the GPIO setting service, the time synchronization with NTP/chrony and the server with the testbed database - were tested, evaluated and finally proved to be well within the requirements. With the GPIO setting service, events can be set accurately with less than $500\mu s$ time difference. The local time of each observer is accurate and creates a common timebase for all observers as a side effect.
The accuracy of all so far implemented parts of the observer proves that the approach of a testbed with a Gumstix as a strong observer is the right way to go. Beyond this master thesis the completion of the testbed will require the dedication of more hardware (observers, target-observer-interface boards, server) and manpower to complete all services and bring the testbed to a state which allows for the start of productive use of the testbed.

Some design and implementation work remains to be done:

- Implementation of remaining services.
- Upgrade of the backbone testbed to run on wireless ethernet.
- Design and implementation of testbed server which will act as time synchronization server, database host, data analyze and data mining server.
- Enhancement of the number of supported target platforms.
- Evaluation of scalability.
- Partitioning of the testbed into subsystems if needed.
- User and access management.
- Security concept and implementation to protect observer from external threats such as: attacks over ethernet, theft ob testbed parts if deployed outside, ...

Moreover, long-term tests with numerous observers involved would allow for further evaluation of the testbed and the implemented services.

Bibliography

[1] Matthias Woehrle. *Proposal for Master Thesis: Wireless Sensor network Testbed 2.0: A new service oriented architecture.* TIK, ETH Zurich, September 2008.

[2] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: ultra-low power data gathering in sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 450–459, New York, NY, USA, 2007. ACM.

[3] SOSUS, sound surveillance system. http://www.globalsecurity.org/intell/systems/sosus.htm. [Online; accessed March 23, 2009].

[4] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[5] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. PermaSense: investigating permafrost with a WSN in the Swiss Alps. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 8–12, New York, NY, USA, 2007. ACM.

[6] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63, New York, NY, USA, 2005. ACM.

[7] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 2006.

[8] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[9] H. Baldus, K. Klabunde, and G. Muesch. Reliable set-up of medical body-sensor networks. In *Proceedings of EWSN 2004: European Conference on Wireless Sensor Networks*, 2004.

[10] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Comput. Netw.*, 46(5):605–634, 2004.

[11] Ivan Stoianov, Lama Nachman, Sam Madden, and Timur Tokmouline. PIPENET: A wireless sensor network for pipeline monitoring. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 264–273, New York, NY, USA, 2007. ACM.

[12] Kay Römer and Friedemann Mattern. The design space of wireless sensor networks. *Wireless Communications, IEEE*, 11(6):54–61, December 2004.

[13] ns-2. http://nsnam.isi.edu/nsnam/index.php. [Online; accessed October 21, 2008].

[14] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on parallel and distributed simulation*, pages 154–161, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *SenSys '03: Proceedings of the 1st international conference on embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[16] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network - a toolkit for the development of WSNs. In *EWSN '07: Proceedings of the 4th European Workshop on Sensor Networks*, pages 195–211. Springer, 2007.

[17] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt 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.

[18] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *REALMAN '06: Proceedings of the 2nd international workshop on multi-hop ad hoc networks: from theory to reality*, pages 63–70, New York, NY, USA, 2006. ACM.

[19] Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. Kansei: a testbed for sensing at scale. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 399–406, New York, NY, USA, 2006. ACM.

[20] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping wireless sensor network applications with BTnodes. In *1st European Workshop on Wireless Sensor Networks (EWSN 2004)*, pages 323–338, January 2004.

[21] I. Haratcherev, G. Halkes, T. Parker, O. Visser, and K. Langendoen. Powerbench: A scalable testbed infrastructure for benchmarking power consumption. In *Int. Workshop on Sensor Network Engineering (IWSNE)*, 2008.

[22] Intel Corporation. *Intel® PXA270 Processor. Electrical, Mechanical, and Thermal Specification*, April 2005.

[23] David Mills. *Network Time Protocol (Version 3) Specification, Implementation.* RFC Editor, United States, 1992.

[24] Matthias Woehrle. Power testing project, TIK, ETH Zurich.

[25] Niclas Finne, Joakim Eriksson, Nicolas Tsiftes, Thiemo Voigt, Adam Dunkels, and Fredrik Österlind. Sensornet checkpointing: enabling repeatability in testbeds and realism in simulations. In *Proceedings of EWSN 2009: 6th European Conference on Wireless Sensor Networks*, Cork, Ireland, 2009.

[26] Intel Corporation. *Intel® PXA27x Processor Family Developer's Manual*, April 2004.

[27] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM.

[28] Gumstix GPIO event driver. http://docwiki.gumstix.org/GPIO_event. [Online; accessed March 27, 2009].

Appendix A

Database Schematics

Observer

tbl_obs_sysinfo
obs_sysinfo_key: INTEGER
property: VARCHAR
value: VARCHAR
last_change: TIMESTAMP

tbl_obs_gpio_monitor
obs_gpio_monitor_key: INTEGER
pin_number: INTEGER
value: CHAR
timestamp: TIME

tbl_obs_powerprofiling
obs_powerprofiling_key: INTEGER
sample: BLOB
samples_per_blob: INTEGER
timestamp_start: TIME
timestamp_end: TIME

tbl_obs_services
obs_services_key: INTEGER
service: VARCHAR
isActive: BOOL
hasData: BOOL
last_change: TIMESTAMP

tbl_obs_gpio_setting
obs_gpio_setting_key: INTEGER
pin_number: INTEGER
value: CHAR
timestamp_planned: TIME
timestamp_executed: TIME

tbl_obs_logmessages
obs_logmessages_key: INTEGER
logmessage_text: VARCHAR
timestamp: TIME

tbl_obs_errorlog
obs_errorlog_key: INTEGER
service: VARCHAR
errormessage: VARCHAR
timestamp: TIME

Server

tbl_serv_sysinfo
serv_sysinfo_key: INTEGER
property: VARCHAR
value: VARCHAR
last_change: TIMESTAMP

tbl_serv_gpio_monitor
serv_gpio_monitor_key: INTEGER
observer_id: INTEGER
obs_gpio_monitor_key_fk: INTEGER (FK)
pin_number: INTEGER
value: CHAR
timestamp: TIME

tbl_serv_powerprofiling
serv_powerprofiling_key: INTEGER
observer_id: INTEGER
obs_powerprofiling_key_fk: INTEGER (FK)
sample: BLOB
samples_per_blob: INTEGER
timestamp_start: TIME
timestamp_stop: TIME

tbl_serv_errorlog
serv_errorlog_key: INTEGER
observer_id: INTEGER
obs_errorlog_key_fk: INTEGER (FK)
service: VARCHAR
errormessage: VARCHAR
timestamp: TIME

tbl_serv_gpio_setting
serv_gpio_setting_key: INTEGER
observer_id: INTEGER
obs_gpio_setting_key_fk: INTEGER (FK)
pin_number: INTEGER
value: CHAR
timestamp_planned: TIME
timestamp_executed: TIME

tbl_serv_logmessages
serv_logmessages_key: INTEGER
observer_id: INTEGER
obs_logmessages_key_fk: INTEGER (FK)
logmessage_text: VARCHAR
timestamp: TIME

tbl_serv_observer_list
serv_observer_list_key: INTEGER
observer_id: INTEGER
ethernet_address: VARCHAR
status : CHAR
last_change: TIMESTAMP

tbl_serv_test_configuration_observer_list_map
serv_test_configuration_key_fk: INTEGER (FK)
serv_observer_list_key_fk: INTEGER (FK)

tbl_serv_test_configuration
serv_test_configuration_key: INTEGER
title: VARCHAR
description: VARCHAR
last_change: TIMESTAMP

Appendix B

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Engineering and Networks Lab (TIK)

Fall Term 2008

MASTER THESIS

for
Christoph Walser

Supervisor: Matthias Woehrle
Co-Supervisor: Andreas Meier

Start date: 29. September 2008
End date: 7. April 2009

Wireless Sensor Network Testbed 2.0: A new service oriented architecture

Introduction

Wireless Sensor Network (WSN) nodes are small battery-powered platforms usually equipped with a micro controller, a radio module and a sensor. These nodes can, for instance, be deployed in a house to measure data (i.e. temperature) and forward this data to a base station. However, the base station is usually not in communication range with all sensor nodes. This requires that the sensor nodes build an ad-hoc network to forward the data over multiple hops, as illustrated in Figure 1. According to a vision of Stankovic et al. [1], this enables a "seamless integration of computing with the physical world via sensors and actuators".

The battery-powered nodes are not only constrained by the very limited power supply (battery) but also in processing power, memory size and communication possibilities to name a few. These limitations result in various difficulties when implementing a WSN application. For instance, an optimized MAC

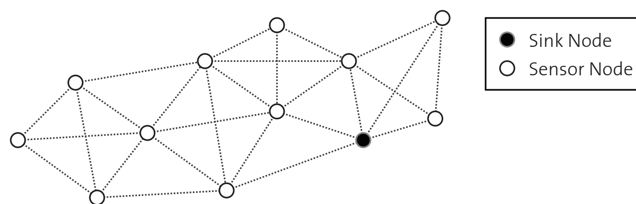


Figure 1: Schematic of a wireless sensor network (WSN): There is usually a dedicated (sink) node acquiring the sensor data gathered by the sensor nodes. Not all sensor nodes are in direct communication range with the node, requiring them to form a multi-hop ad-hoc network.

protocol is required to duty cycle the node's radio, whereas the routing protocol has to deal with the very limited memory and processing capabilities. Especially the combination of low-power operation and wireless communication seems to be a crux for the robustness and reliability of WSNs. This combination results in unpredictable communication channels (links) between the nodes due to reasons of interference and fading. Various research groups analyzed this behavior [2, 3] and showed that the quality of different links varies greatly and that some links show a very unpredictable and not deterministic behavior.

In order to arrive at a functioning system at deployment time, several test platforms have been proposed such as simulators [4, 5], emulators [6] or various testbeds [7, 8, 9]. While other test platforms abstract away from the intricacies of the hardware, a testbed provides code execution on real hardware typically placed in an environment which closely resembles the deployment region.

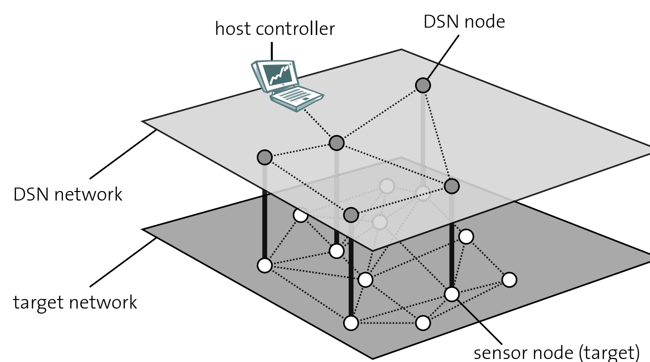


Figure 2: The target nodes are connected to the node of the 'Deployment Support Network' (DSN) with a short wire. This secondary (wireless) network provides a flexible access to the target nodes, in particular it allows updating the application code, logging data from and sending commands to the targets.

Testbeds are used for different kinds of test:

- Integration testing, to check whether the application adheres to the specification,
- Performance testing, to check whether the application's performance is acceptable under differing environmental conditions,
- Regression testing, to check wether previously found problems do not exist in newer software versions,
- Stress testing, to determine under which deviating environmental conditions the system may fail,
- etc.

While current testbeds have several features, which allow for superior testing than on other platforms, they often lack features indispensable for general testing tasks: tight synchronization, reproducibility among others.

Problem Definition

The main goal of this thesis is to design, evaluate and analyze a service oriented testbed architecture based on a Gumstix Verdex [10] platform. The Gumstix platform includes a powerful processor and the possibility to connect via ethernet or WLAN to the back-end architecture.

In order to reach this goal, the project should proceed according the following steps:

1. The student should write a project plan and identify its milestones (thematic and time wise). In particular there should be enough room for the final presentation and the report.
2. The student should study related work in the area of WSNs, focusing on testbeds [7, 8, 9] and testing [11, 12, 13] and WSN development and its intricacies [14, 15, 16]. The student should also look out for further related work in this area. The results of this literature research should be written down as a first chapter of the report.
3. The first task is to collect requirements for distributed sensor node testing and present them in a concise and detailed manner. The Gumstix platform is in turn evaluated based on these criteria on its applicability in WSN testing.
4. The second task is designing and implementing a resource-efficient logging utility for the sensor node itself with minimal probing effect. Staring point for this work is a previous semester thesis that presents a general overview and interfaces as well as the work on EnviroLog [17] and NodeMD [18]. An evaluation of buffer size versus log size and log frequency versus local logging reliability should be performed based on several selected testcases.
5. The third task is to implement a method for programming a node from the Gumstix module with the possibility to retrieve and apply RAM state via the bootlader. Possible issues are that the bootloader also resides in RAM and may override program state, the requirement of idle peripherals and the initialization of peripherals before applying a RAM image.
6. The fourth task is data collection and synchronization on the observer. For once a a method for transferring information from and to the Gumstix module from a local development host as with the DSN Logger and time synchronization of the observer using NTP or other protocols such as proposed in [19].
7. The last task is to apply the tools provided on real test, tests for the PermaSense [20] and the Harvester [21] project using the tools and mechanisms provided above.
8. This is the set of tasks that must be accomplished. Further work is performed based on preferences and strengths of the student.

Organization

- Duration of the Work:
  This Master Thesis starts 29. September 2009 and has to be finished no later than 30. April 2009.
- Project Plan:
  A project plan with its milestones is held and updated continuously. Unforeseen difficulties that change the project plan have to be documented and should be discussed with the supervisors.
- Weekly Meetings/Reports:
  In regular (weekly) meetings with the supervisors, the current state of the work, potential difficulties as well as future directions are discussed. The day before the weekly meeting a brief status report should be sent to the supervisors commenting on these issues, in order to allow an adequate meeting preparation for the student and the supervisors.
- Research Journal:
  The work's progress is written down in a research journal that is handed in to the supervisor at the end of the project.
- Progress Reports:
  Every Month a short report of approximately 5 pages summarizes the progress, status and next steps for the thesis.

- Beginners Presentation:
  Approximately two to three weeks after the start you will shortly present the objectives of the work as well as some background on the topic. The presentation should be no longer than 5 minutes and consist of maximally two slides.
- Final Presentation:
  By the end of the project, you will present the achieved result. The presentation should not exceed 20 minutes.
- Documentation:
  At the end of the project, no later than 24. April 2009, you will have to hand in a written report. Together with the system implementation/software this report is the main outcome of the project. Document your work accurately. Additionally make sure to comment your code extensively, allowing a follow-up project.
- Evaluation of the work:
  The criteria for grading the work are described in [22].
- Finishing up:
  The required resources should be cleaned up and handed back in.

References

[1] J. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems.," IEEE Computer, vol. 38, no. 11, pp. 23–31, 2005.

[2] J. Zhao and R. Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in First Int'l Workshop on Embedded Software (EMSOFT 2001), pp. 1–13, 2003.

[3] N. Reijers, G. Halkes, and K. Langendoen, "Link Layer Measurements in Sensor Networks," in Proc. 1st Int'l Conf. on Mobile Ad-hoc and Sensor Systems (MASS '04), Oct. 2004.

[4] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003), pp. 126–137, Nov. 2003.

[5] L. Girod, N. Ramanathan, J. Elson, T. Stathopoulos, M. Lukac, and D. Estrin, "Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks," ACM Trans. Sen. Netw., vol. 3, no. 3, p. 13, 2007.

[6] B. L. Titzer, D. K. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05), p. 67, 2005.

[7] G. Werner-Allen, P. Swieskowski, and M. Welsh, "MoteLab: A wireless sensor network testbed," in Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05), pp. 483–488, Apr. 2005.

[8] V. Handziski, A. Koepke, A. Willig, and A. Wolisz, "Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in Proc. 2nd international workshop on Multi-hop ad hoc networks: from theory to reality (REALMAN '06), (New York, NY, USA), pp. 63–70, ACM Press, 2006.

[9] 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 Proc. 4th European Workshop on Sensor Networks (EWSN 2007), pp. 195–211, 2007.

[10] gumstix inc., "gumstix - dream, design, deliver," September 2008.

[11] Rincon Research Corporation, "Tinyos 2.x automated unit testing / tunit." http://www.lavalampmotemasters.com/, May 2008.

[12] M. Woehrle, C. Plessl, J. Beutel, and L. Thiele, “Increasing the reliability of wireless sensor networks with a distributed testing framework,” in Proc. 4th IEEE Workshop on Embedded Networked Sensors (EmNetS-IV), pp. 93–97, ACM, 2007.

[13] M. Woehrle, J. Beutel, and L. Thiele, “Wireless sensor networks testing and validation,” in Handbook of Embedded Systems, To appear.

[14] P. Levis, “Tinyos programming,” June 2006.

[15] J. Choi, J. Lee, M. Wachs, and P. Levis, “Opening the sensornet black box,” Tech. Rep. SING-06-03, Stanford Information Networks Group, Stanford University, CA, 2006.

[16] K. Langendoen, A. Baggio, and O. Visser, “Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture,” in Proc. 20th Int’l Parallel and Distributed Processing Symposium (IPDPS 2006), pp. 8–15, 2006.

[17] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, “Achieving repeatability of asynchronous events in wireless sensor networks with envirolog,” in INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings, pp. 1–14, Apr. 2006.

[18] V. Krunic, E. Trumpler, and R. Han, “Nodemd: diagnosing node-level faults in remote wireless sensor systems,” in MobiSys ’07: Proceedings of the 5th international conference on Mobile systems, applications and services, (New York, NY, USA), pp. 43–56, ACM, 2007.

[19] P. Blum, Guaranteed Time Synchronization in Wireless and Ad Hoc Networks. PhD thesis, Dept. Information Technology and Electrical Engineering, ETH Zürich, Switzerland, Nov. 2004.

[20] SwissExperiment, “Permasense:home - swissexperiment,” September 2008.

[21] Computer Engineering and Networks Lab - ETH Zürich, “Harvester in tinyos 2 contrib,” 2008.

[22] TIK, “Notengebung bei Studien- und Diplomarbeiten.” Computer Engineering and Networks Lab, ETH Zürich, Switzerland, May 1998.

Appendix C

Work Schedule

Project plan master thesis Christoph Walser

Version: 2009/02/17 --> **Milestone**

	2008												2009												
Week	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Study related work													Xmas - Break (5 days)														
Introductions by team members																											
Set up tools, familiarize with used HW and SW																											
Assemble and get to work GumStix and TinyNode																											
Log data from TinyNode over GumStix on PC																											
Experiment design and preparation																											
Data logging utility																											
Database performance evaluation																											
Event logging utility																											
Database analysis, implementation																											
Time synchronisation accuracy of NTP																											
Time response accuracy of GPIO																											
Implement Services																											
Test all implementations in overall test scenario																											
Refine and adjust implementations																											
Write report																											
Prepare/give presentation																											

Statement regarding plagiarism when submitting written work at ETH Zurich

By signing this statement, I affirm that I have read the information notice on plagiarism, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Information notice on plagiarism:
http://www.ethz.ch/students/semester/plagiarism_s_en.pdf

________________________	____________________________________
place and date	signature