Wireless Yet Reliable Patient Notification System: GUI and Back-end

Semester Thesis

Reto Da Forno

February 18, 2013

Supervisor: **Marco Zimmerling**
Co-Supervisors: **Federico Ferrari, Roman Lim, Olga Saukh**

Computer Engineering and Networks Laboratory (TIK), ETH Zurich

Abstract

People suffering from muscular dystrophy are often wheelchair-bound and unable to move independently. It's a necessity to enable them to contact their caretaker during the night. It would be of great use to have a mobile notification system which could easily be installed, used for a few weeks and removed, wherever no patient notification system is available. Such a system will be realized with wireless sensor nodes and a recently developed communication scheme that provides certain guarantees on wireless communication. In this semester project, we developed a comfortable user interface (GUI) and back-end for the system to collect, store and display events as well as notify the patients. The application runs on any recent personal computer and enables monitoring and operation of the notification system in real time.

Contents

1 Introduction **3**
- 1.1 Problem and Motivation 3
- 1.2 Thesis Contributions 3
- 1.3 Thesis Outline 4

2 Background **5**
- 2.1 Application Scenario 5
- 2.2 Wireless sensor networks 5
- 2.3 Low-power wireless bus 6

3 Requirements **7**
- 3.1 Functional Requirements 7
- 3.2 Non-functional requirements 8

4 Design **9**
- 4.1 Assumptions 9
- 4.2 Components and Interfaces 9
- 4.3 Server 10
- 4.4 Graphical user interface 10
- 4.5 Data structures 11
- 4.6 Database scheme 12
- 4.7 Serial Port Communication 12
- 4.8 Data Flow 13
- 4.9 Extension by additional features 13

5 Implementation **15**
- 5.1 Technologies 15
- 5.2 Main application 15

6 Testing and Results **18**
- 6.1 Methods 18
- 6.2 Scenarios and Coverage 19
- 6.3 Performance 19
- 6.4 Platform and versions 20

7 Conclusion and Future Work **21**

A Adding new features **22**

B User Guide **23**
B.1 Files and folder structure 23
B.2 How to use the application 23
B.3 Using the GUI 26
B.4 Change the basic configuration 27
B.5 Configure SMS support 28

C Program flow **29**
C.1 Python application 29
C.2 Javascript application 31

D Debugging **32**

Bibliography **33**

Chapter 1

Introduction

1.1 Problem and Motivation

People suffering from muscular dystrophy often cannot move without help. Many of them need 24 hour assistance. The Schweizerische Muskelgesellschaft organizes summer camps for those people (mostly kids and young adults) every year at different locations. In the past years, the organizers of the summer camp had used a custom system built out of home-made hardware. However, this system must be installed first because it's based on wired connections. The cabling involves a lot of work and implies very limited mobility and scalability. The high voltage which was required to drive the system resulted in high power consumption and occasionally led to unpleasant side effects (e.g. occasional electric shocks). The current system is far from ideal and even showed to be insufficient.

Baby monitors have not been considered as an alternative as they violate the privacy, which is especially problematic when used for adults. Mobile phones might be an alternative in the present age, but would require more effort from the patients instead of just pressing a button, not to mention that some of them may not be able to operate a standard mobile phone. Special devices for handicapped or elderly people would certainly work, but are often ineffective and overpriced. Such devices would cause relatively high cost because one device for each participant will be needed. Furthermore, mobile communication – if used frequently – causes significant additional expenses.

The camp organizers are looking for a new notification system which is easy-to-install, easy-to-use, flexible, reliable and affordable at the same time. Currently, there seems to be no out-of-the-box solution that satisfies all these requirements.

1.2 Thesis Contributions

The new notification system will be realized on the Tmote Sky platform, a hardware commonly used in wireless sensor networks. A recently developed communication scheme proposed by Ferrari et al.[1] allows for multi-hop communication and provides certain guarantees on wireless communication. A sink node will be connected to a computer and represent the interface between the

wireless sensor network and the operator of the system. The contribution of this semester thesis to the wireless notification system is to provide a comfortable user interface (GUI) and back-end to collect, store and display events as well as notify the patients. The program which we developed during this semester project can be used on any recent general purpose computer which has a USB port.

1.3 Thesis Outline

We are going to provide some background information in the next chapter and list the requirements for the program in the third chapter. The section Design is targeted to serve as a high-level overview. Chapter five is dedicated to the implementation and six will reveal some details on how the system was tested. In the last chapter, we discuss the results and give a brief outlook. Additional information as well as a user guide for the developed application can be found in the appendix.

Chapter 2

Background

2.1 Application Scenario

The Schweizerische Muskelgesellschaft organizes annual summer camps for children and adults who are suffering from muscular dystrophy. These people have a very limited ability to move and therefore need assistance throughout the day. During the summer camps, a notification system is needed to enable the participants to call for help during the night, e.g. by pressing a button. Those camps take place at different locations without any pre-installed notification system as it can be found for instance in a hospital. Sometimes, the camp facilities architecturally prohibit to have the caretaker's room in the same building. The camp organizers are looking for an easy-to-install, easy-to-use, flexible and highly reliable notification system. Furthermore, it must be low-cost to be affordable by the non-profit organization. In the typical use case, between 10 and 20 nodes with 1 to 3 Buttons attached to each of them will be needed. This means, every patient has its own button but up to 3 patients, who are physically close to each other, might share the same sensor node. The spacial extent of the deployment area is usually no more than 100 meters in diameter. Although wired systems as used in hospitals are very reliable, they also involve lots of cabling and a time consuming installation. Imposed by the given constraints, the use of wireless sensor networks (WSN) is a suitable choice. Of course, a wireless system would also need to be installed, but with a much lower effort. Despite the lossy nature of wireless channels, communication can be made highly reliable by using appropriate protocols such as the Low-power wireless bus (LWB).

2.2 Wireless sensor networks

"Wireless sensor networks consist of spatially distributed sensors, often used to monitor environmental data." [2] These sensors are called nodes and are somehow connected to each other through a wireless link. Nodes are usually very small, cheap and designed to operate autonomously for years. All measurements collected by the sensor nodes are forwarded to a base station, which stores all data and – if necessary – uploads the data to the Internet over the cellular network or a wired connection. The sensor nodes themselves have high

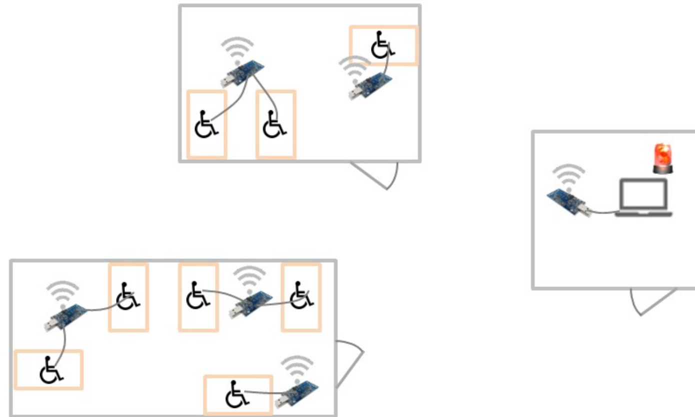

Figure 2.1: exemplary layout of a camp site

power constraints and therefore very limited computational power and wireless transmission range to keep the consumption as low as possible. Special communication protocols are needed to meet this requirement. Usually, the nodes do not communicate directly with the base station (sometimes also called gateway sensor node or sink). Instead, a multi-hop topology is used to forward the messages to their destination. Wireless sensor networks are an active and potential research area with numerous applications and will become even more important in the future.

2.3 Low-power wireless bus

To provide reliable wireless communication and yet conserve battery power, special protocols are needed. The Low-Power Wireless Bus is a communication protocol that supports several traffic patterns and "turns a multi-hop low-power wireless network into an infrastructure similar to a shared bus, where all nodes are potential receivers of all data." [1] This is achieved by network floods coordinated by a global schedule. The protocol is more resilient to link changes than comparable competitors because no topology-dependent state is kept. Therefore, interference and even node failures affect the performance only marginally. Each sensor node is assigned to a time frame for sending data. The LWB protocol gives some guarantees that a message will be delivered within a certain time span. If a message does not arrive, it can be assumed that the sensor node has moved away or has stopped operating properly. Further details on how the Low-Power Wireless Bus works can be found in the corresponding paper proposed by Ferrari et al. [1] In the meantime, the version of LWB described in the paper has been improved to further increase the probability of a successful transmission.

Chapter 3

Requirements

The goal of this semester project is to develop a lightweight GUI which acts as an interface between the wireless sensor network and the operator of the system. A sink node will be connected to a portable personal computer via USB. The computer takes the role of a base station and collects, stores and displays all data from the wireless sensor network. In this semester project, the software components for the base station will be developed. This includes the implementation of a serial port I/O handler, a lightweight database for event logging and a user interface, where the whole system is monitored and all events are displayed to the operator.

The following requirements are related to our part of the project, namely the GUI and the back-end. Therefore, previously mentioned requirements such as low cost aren't listed here. Most requirements were discussed with the people from the Muskelgesellschaft to build the application as adequately as possible to their needs.

3.1 Functional Requirements

- The operator of the notification system must be able to assign a location to each node as well as a patient to each button of a node.
- A list of all sensor nodes within the WSN and the patients which are assigned to them must be displayed.
- The status of all nodes (active or lost) and all buttons (pressed or not) attached to them must be displayed.
- The system must automatically detect and mark "lost" nodes. A node is called "lost" if it doesn't send "alive" status messages any more, either because it stopped working properly or the status message cannot be transmitted successfully due to interference or an out-of-range situation.
- When a button is pressed by a patient, the red LED on the corresponding node will turn on and an alarm message shall be sent to the GUI. The alarm must be announced visually and audibly to the operator (probably with increasing alarm sound volume).

- As soon as the alarm message is received by the application, an acknowledgement will be sent back to the sensor node that triggered the alarm message and a yellow LED will be turned on.
- When the operator confirms the alarm by pressing the corresponding button in the GUI, another acknowledgement shall be sent back to the sensor node. This will enable the green LED on that node, indicating to the patient that the operator has received the alarm message and a caretaker is coming.
- An alarm shall only be triggered once until its confirmation and a small time frame afterwards, even if the patient presses the button several times.
- An SMS shall be sent if an alarm is not confirmed by the operator within a certain amount of time.
- There should be an option in the GUI to import a CSV file with a list of all patients and the nodes and buttons to which they are assigned. Whenever a new list of patients is imported, the old list should be saved in a history table to be able to trace all events back.
- All important settings should be changeable from the GUI.
- Some basic statistics (such as average response time or number of alarms in the past hour) should be displayed.

3.2 Non-functional requirements

- The GUI must be easy-to-use, i.e. simple and clear.
- The application will run on a general purpose computer, but must be platform independent.
- Recent technology that is widely supported and will still be available in 10 years shall be used.
- The sink sensor node will be connected to the computer via USB, i.e. the application must communicate over the serial port interface.
- An alarm must be clearly visible at any time (no occluding menus or popups).
- An alarm must be displayed in the GUI within a few seconds after a button was pressed. It doesn't need to be real time, but within a reasonably small time frame (e.g. 10 seconds).
- The transmission between the sink and the GUI must be reliable.
- The task must be completed within 14 weeks.

Chapter 4

Design

Each patient gets a button which is attached to a sensor node. Sometimes, several patients will share the same node and therefore, one node might have several buttons. Every few seconds, each sensor node sends a status message to the sink to show that it is still there. The sink node itself is connected to a personal computer via a serial link (USB). All status messages are forwarded to the GUI where they can be observed by the system operator. Whenever a node fails or gets out of range, it will be notices by the system and the nodes status will be set to "lost". If a patient presses a button, an alarm message is sent to the sink node, transmitted to the computer, stored in the database and forwarded to the GUI. The whole transmission won't take longer than a few seconds. When the alarm message arrives at the GUI, it is indicated visually and audibly. As soon as the operator has confirmed the alarm, an acknowledgement packet is sent back through the wireless sensor network to the node which generated the alarm message. A green LED turns on which tells the patient that the alarm has been delivered and help is coming.

4.1 Assumptions

The design of the software is based on the reliability of the Low-Power Wireless Bus. We assume that a message arrives at the sink node with very high probability. To simplify the data flow model, we adopt a success rate of 100 percent in both directions, i.e. we don't have to bother about packet losses between the sink node and all other nodes within the wireless sensor network.

4.2 Components and Interfaces

Basically, there are three components: an application which handles the serial port I/O, a database and a graphical user interface (see fig. 4.1). The interface between the sink node and the computer is a USB link and therefore involves serial port communication. The interface between the database and the main application will be provided by the chosen database application (see chapter 5). The connection between the main application and the GUI will involve some thread-to-thread communication which can be accomplished by many different approaches, including message passing or shared memory. From now on,

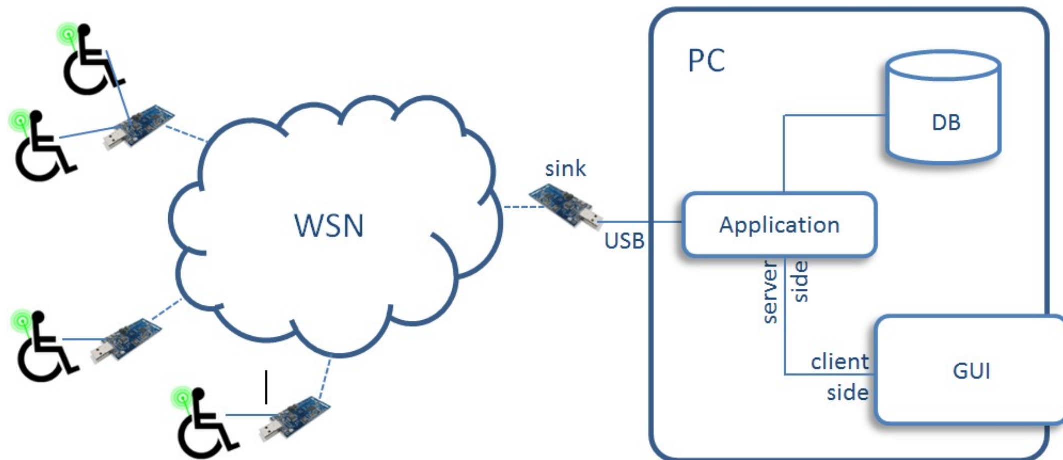

Figure 4.1: system overview

we will refer to this interface as the connection between a server (application side) and a client (GUI).

4.3 Server

The server is a part of the application and acts as an interface to the GUI. To make the application smooth and allow multiple simultaneous data exchanges, this part of the application should run in a separate thread. That's why the communication between the main thread and the server needs to be ensured by two FIFO queues, one to push data to and one to read data from the server. The client (GUI) can request status updates, whereupon the server pops an element form the input message queue and send it to the client. If there's no element in the queue, the server waits until an element becomes available or the timeout is reached. If the latter occurs, the last known status is returned to the client. I.e. the client will receive a status message from the server periodically, which indicates that the server is still running flawlessly.

4.4 Graphical user interface

The following elements must be displayed in the user interface:

- the status of all connected nodes and attached buttons
- the overall system status
- some statistics (e.g. number of alarms in past hour)
- an option to change the settings as well as to import and edit the list of patients

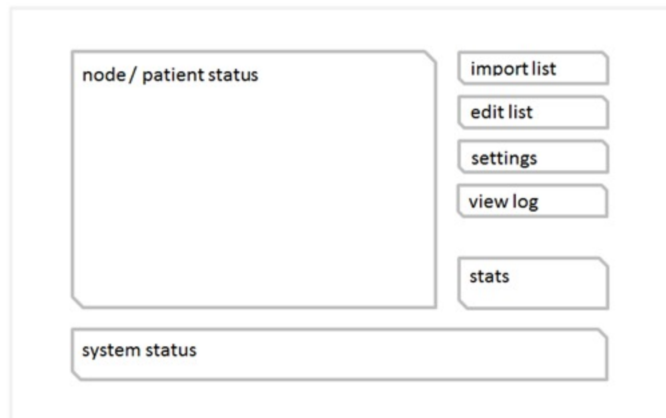

Figure 4.2: first sketch of the GUI

If the system operator has not previously selected a serial port, a list of connected USB devices will be shown and the right device can be chosen. Initially, the operator should upload a CSV file containing a list of all patients and the assigned node and button IDs. Whenever an alarm is delivered to the user interface, the corresponding node will be highlighted in red. Additionally, an alarm sound will be played and the border around the node list box will start to blink. If an alarm is not confirmed within a minute (or a custom timeout value), an SMS will be sent to up to three specified mobile phone numbers. In this case, a blinking mobile phone will appear in the GUI. If a node disappears, i.e. does not send a status message within a specified amount of time, it will be highlighted in yellow.

4.5 Data structures

All packets received through the serial link will be parsed by the application and forwarded accordingly depending on their content. A list of all sensor nodes in the WSN is stored locally. All necessary information including the status and attached buttons is kept in this data structure. Only if the status of a node changes, the status of all nodes in the local list will be passed to the server in the form of a unified status message by adding it to the input FIFO queue. This functionality prevents the exchange of unnecessary data between the server and the client (GUI). The data format for elements in the input message FIFO queue is as follows:

```
{ [type], [message] }
```

The following example is a unified status message indicating that node 0, 1 and 2 are OK. The second example shows how an alarm message looks like (button 1 on node 0 was pressed).

```
{ "S", "0 (1)|1 (1)|2 (1)" }
```

```
{ "B", "0-1" }
```

There is no specific reason for the choice of the format other than readability.

table	fields (data types)
patients	id (integer UNIQUE AUTO), name (string), room (string), buttonid (string)
patientshistory	id (integer UNIQUE AUTO), name (string), room (string), buttonid (string), timestamp (string)
events	id (integer UNIQUE AUTO), buttonid (string), time (string), confirmed (string)
config	key (string UNIQUE), value (string)

Table 4.1: database tables

Framing Byte 0x7e	Seq. No. (2 Bytes)	Payload	CRC16 (2 Bytes)	Framing Byte 0x7e

Table 4.2: packet format for serial port communication

4.6 Database scheme

According to the feature list, we need to log all alarm events into a database and a list of all patients must be kept, too. Furthermore, a table for the configuration values is needed. As the list of patients might be updated by the system operator by a CSV file import, the old entries in the table "patients" must be copied to an additional history table and marked with a time stamp. This enables to trace all events back and assign all past alarms to a patient. Table 4.1 lists all database tables, the included fields and their data types.

Note that the field "buttonid" holds both, the node and the button ID, i.e. an alarm from the button 1 on node 0 would result in a value of "0-1" for buttonid. In addition to the ID, each alarm event is logged along with two time stamps: the first one tells when the alarm occurred and the second one when it was confirmed. For simplicity, we chose the data type string for the time stamps, but it could as well be stored as an integer or a floating point number.

4.7 Serial Port Communication

Data is exchanged byte by byte over the serial port without a built-in verification functionality to check whether the data has been transmitted successfully or not. Therefore, a checksum calculation is necessary. We decided to use the cyclic redundancy check (CRC16) [3] to detect transmission errors. It is fast, easy to implement and provides a sufficiently high error detection rate for small packets. Sending the CRC checksum along with the message in plain text would work just fine for our application, but might lead to problems when transferring arbitrary (binary) data. Inspired by the implementation in TinyOS [4], we chose a more sophisticated approach with framing and encoding. TinyOS [5] has adopted some aspects of the Point-to-Point Protocol (PPP) [6], which is the successor of SLIP [7] and based on HDLC [8]. Today, PPP is a widely used standard protocol for the link between the internet service provider and its customers. We decided neither to use PPP nor the implementation of TinyOS, but instead a modification thereof to prevent unnecessary overhead. Table 4.2 illustrates the packet format.

In our application, the payload is an ASCII string, but could as well be arbitrary data. To ensure that the packet is correctly recognized at the receiver, all framing bytes (0x7e) between the frame delimiters must be escaped. In our case, this is done by inserting an additional byte (0x7d) before every occurrence of a framing byte within the packet. All escape bytes themselves which should not be regarded as a control byte but rather as a general data byte must also be escaped. Using this encoding, packets with arbitrary content will be correctly detected by its frame delimiters.

4.8 Data Flow

The time-space-diagram in fig.4.3 shows an example of a possible data flow between the components for a very simple setup with one sensor node and one sink node. We assume the initial system state is S (1,1), i.e. node 1 is working properly, and there are no pending alarms. The sink node periodically receives status messages from all sensor nodes within the WSN and forwards them to the computer. In our example, there's just one node (with ID 1). The period is designated as status message interval in the figure. The first received status message is S (1,0), which indicates that the sink node has not received a status message from node 1. This new status is forwarded to the GUI. The next status message is still the same, that's why there's no need to forward this message to the user interface. Whenever the CRC checksum doesn't match, the packet is rejected and the application waits for the retransmission. Alarm messages are treated the same way. An alarm triggered from the same button is only forwarded to the GUI once, but is retransmitted if it is not confirmed by the operator within a predefined amount of time (default: 10 seconds). This is a precaution measure to make sure that the alarm is displayed on the user interface. As soon as the alarm is confirmed, an acknowledgement is sent back to the sensor node with ID 1 and a green LED will turn on. Another failure which might occur is that no status message is received from the sink node within a specified amount of time (designated by the message timeout in the figure). In this case, the application generates a status message S (1,0) and sends it to the GUI to indicate a possible node loss.

4.9 Extension by additional features

The system is designed to enable easy integration of additional features. The Interfaces allow to exchange arbitrary data, i.e. they do not limit in any way. It's only necessary to slightly adjust the components such that they can handle the new messages. A specific example on how to integrate a new functionality can be found in the appendix.

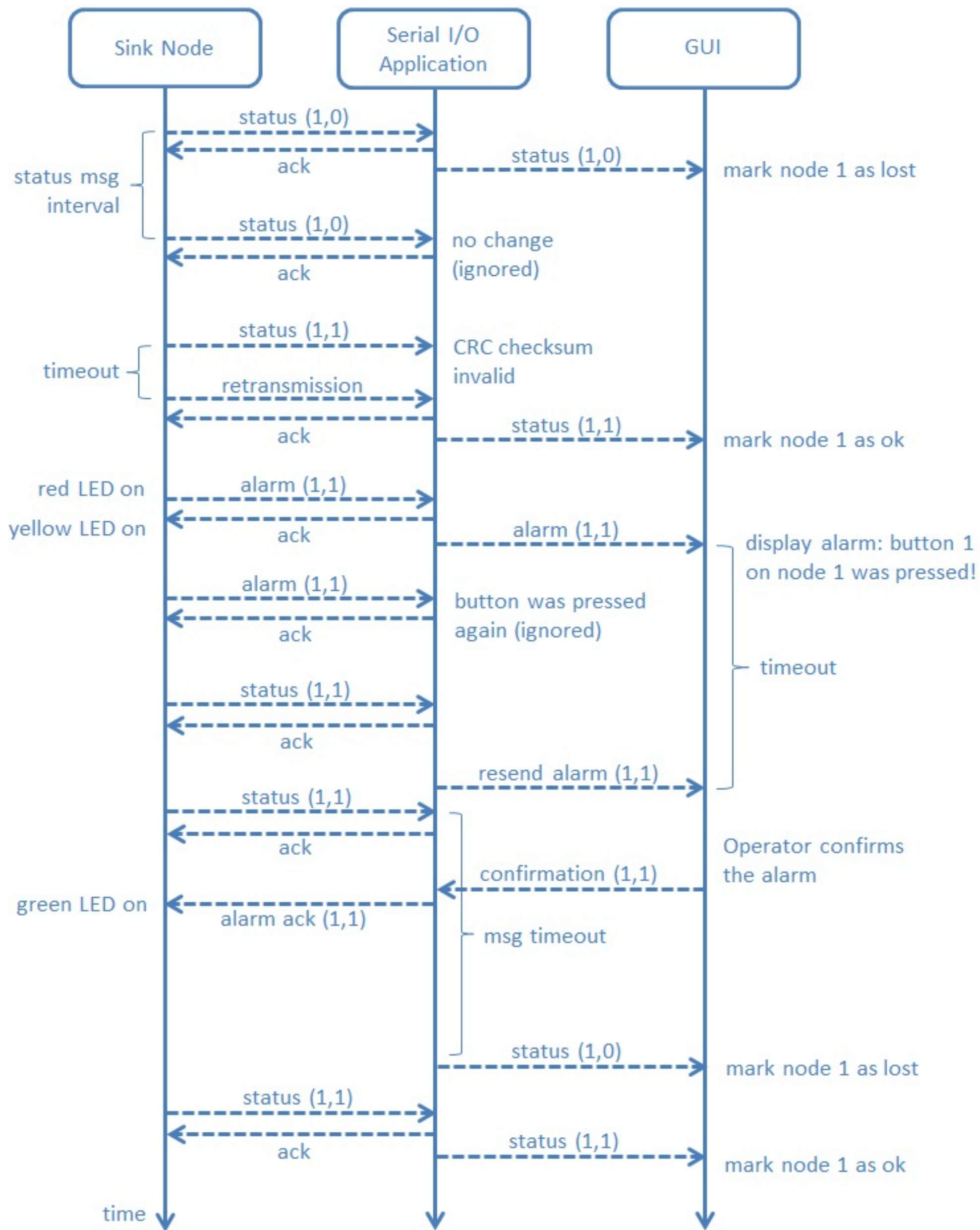

Figure 4.3: data flow between the components

Chapter 5

Implementation

5.1 Technologies

We have implemented the required components with state-of-the-art technologies. Python was chosen for the serial port I/O and Javascript to build the user interface in the form of a webpage. Those programming languages are platform independent, widespread, easy-to-use and yet powerful enough to meet the requirements. We preferred Python to Java because of the native support for serial port I/O enabled by the plugin PySerial [9]. To keep the communication between the Python program and the user interface (GUI) simple, we decided to use a client-server infrastructure. The GUI will be displayed in an arbitrary web browser by connecting to the local HTTP server. It is basically just an html page. Furthermore, XMLHttpRequests will be used for communication between the client and the server to prevent unnecessary reloads of the webpage. "XMLHttpRequest (XHR) is an API available in web browser scripting languages such as JavaScript." [10] It is a widely used tool for AJAX (Asynchronous JavaScript and XML) applications for browser-server communication to build responsive and dynamic web applications. As already mentioned, this technique enables the web application to retrieve new data from the server without reloading the whole page.

In addition, we decided to used jQuery [11], a widely used library for developing Javascript applications. The use of selectors and event handlers simplifies the code and enables separation of structure (HTML elements) and behaviour (functionality), just as structure and presentation (formatting) are separated by Cascading Style Sheets (CSS). Furthermore, the programmer doesn't have to cope with browser dependent calls because jQuery does all the necessary work. "jQuery is a lot about making code shorter and therefore easier to read and maintain." [11]

As for the database, SQLite [12] seems to be the obvious choice because it is simple, lightweight and already integrated in Python.

5.2 Main application

The main application written in Python consists of two basic components:

- The worker thread which handles all the serial port I/O.

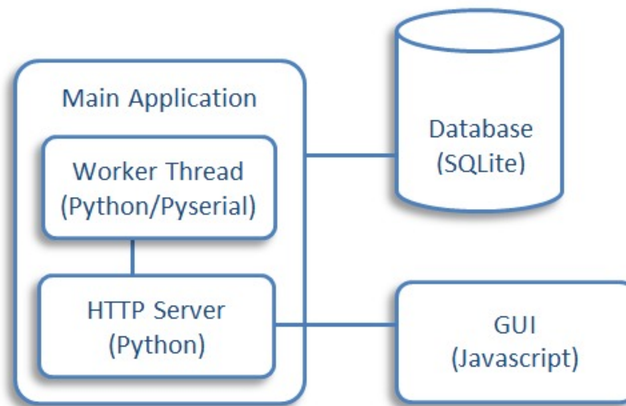

Figure 5.1: components and interfaces

- The HTTP server which enables communication between the Python program and the GUI (client).

Only the worker thread has access to the serial port, collects all data and passes it to the HTTP server. This is achieved by a thread-safe message queue.

The database is opened once at the beginning and kept open until the program is shut down. To enable database access for all threads, it must be protected by some mechanism. The database can be accessed over a globally defined object and is guarded by a mutex to ensure that only one thread at a time reads from or writes to the it. Special attention must also be paid to all other global variables as they might be accesses simultaneously by multiple threads. Before a thread writes to a global variable, it must acquire a lock to make sure mutual exclusion is guaranteed in the critical section.

All packets read from the serial port are parsed by the worker thread. After the checksum has been verified and an acknowledgement has been sent back to the sink node, the received message must be interpreted. There are two valid message types with the following formatting:

- Status messages: `"S,[node],[status]"`
- Alarm messages: `"B,[node],[button]"`

If the received message is not formatted as expected, it will be ignored.

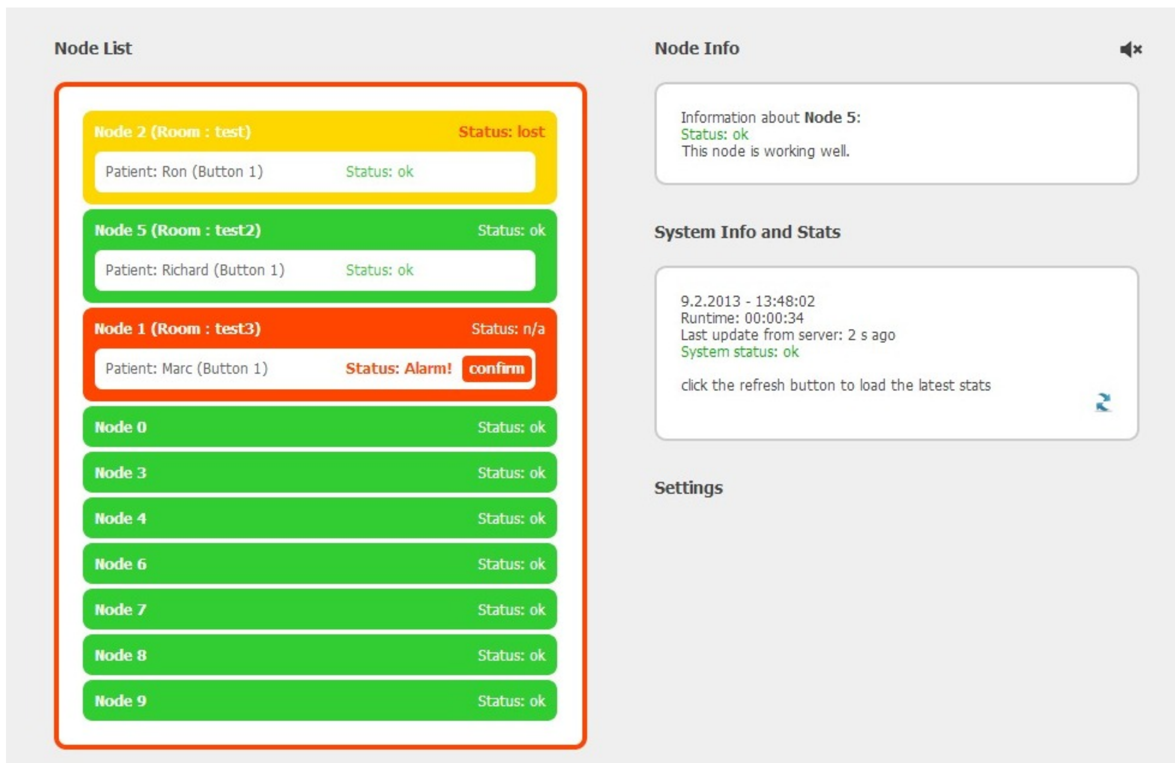

Figure 5.2: the final GUI

Chapter 6

Testing and Results

6.1 Methods

We have tested the application with a real device (Tmote Sky) which outputs status messages for 20 nodes every second, whereof exactly one was marked as lost (status = 0). Additionally, one alarm message was generated every minute. With this setup, a sensor network consisting of 20 nodes and one button attached to each of them was simulated. This first test was mainly to demonstrate the right functioning of the serial port communication. A second test was carried out with an emulator, which can be configured using probabilistic values. This made testing much easier and more flexible. The emulator is basically just a function which is called instead of reading from the serial port. This function returns a status message or an error message. Several parameters can be set for the emulator:

- The number of nodes (default: 10)
- The period: one status message for all nodes is generated in every period (default: 2 seconds)
- The probability of a button to get pressed within one period
- The probability of a node to become unavailable (status 0) within one period
- The probability of a node to fail within one period (no status message received)

The last two points might seem to be the same, but they are not. If a node becomes unavailable, then the sink node is aware of that and sends a message with status = 0 to the computer. If in contrary a node fails and the sink node does not recognize it (e.g. if the node is out of range), then no status message for that node will be received. Therefore, both cases are necessary and must be distinguished. The number of buttons per node is chosen randomly between one and three.

malfunction	countermeasure
The sink node is disconnected unexpectedly or stops operating such that no more data is received.	If no packet is received within a specified time, the problem will be forwarded to the GUI and displayed to the user.
The sink node sends garbage (not correctly formatted packets).	Invalid packets (either CRC checksum wrong or invalid content) will be ignored and are not counted as received packets. Again, if the timeout occurs, it will be visible in the GUI.
Transmission errors occur (bit flips).	Can be detected with high probability using CRC.
An arbitrary node in the wireless sensor network fails or gets out of range.	A list of all nodes is stored on the server. If no status message is received for a node within a certain amount of time, the status will automatically be set to "lost".
The server (Python script) crashes or message forwarding from the server to the client fails.	The server will be restarted automatically whenever an exception occurs. Furthermore, the GUI keeps track of all incoming data from the server. If no data is received within a certain timespan, a warning will be displayed to the user.
The Javascript program crashes.	Will be visible in the GUI either because the clock stops or because no more packets are received from the server.

Table 6.1: possible malfunctions

6.2 Scenarios and Coverage

With both tests, a wide range of scenarios has been covered. Because the sensor network was not yet available at the time we tested the system, a real world scenario could not be examined. There shouldn't be any complications when the fake sink node is replaced by a real system as long as the packet and message format is observed. Also, the test was accomplished with 20 sensor nodes or less according to the intended use of the system. Our tests did not cover scenarios with more than 20 sensor nodes or more than 3 buttons attached to one node. Table 6.1 lists some malfunctions which could occur and the corresponding countermeasures which were taken to minimize the negative impact on the system.

6.3 Performance

As mentioned earlier, the Python program will run on a general purpose computer with sufficient computational power. However, there's no need to strain it more than what is necessary. To reduce the CPU time, a limiter was inserted in the worker thread. This limiter implemented with a sleep statement restricts the maximum number of loop passes per second – and therefore the

number of packets which can be read from the serial port – to 50. We have not observed any performance issues with up to 20 nodes. Everything runs smooth and the GUI is responsive.

6.4 Platform and versions

The application was tested on a Windows 7 machine with Python 2.7.3, Pyserial 2.6 and jQuery 1.8.3. The client side (GUI) was tested on several browsers. It runs flawlessly on Google Chrome 23, Mozilla Firefox 9 and Internet Explorer 9. It should basically work on any browser which has Javascript turned on and supports HTML5. Javascript has somewhat limiting functionality when it comes to reading files. For security reasons, files stored on the local hard disc can generally not be accessed with Javascript. With the introduction of HTML5, this limitation had been removed by file readers which enable file uploads without refreshing the page. Also, playing audio files has been simplified with HTML5. We decided to use HTML5 because of these two advantages.

Chapter 7

Conclusion and Future Work

We were very pleased with the chosen technologies for the implementation. Javascript and Python are both very easy to learn, lightweight and yet powerful in their functionality (see restrictions in appendix D). Other options such as Java or C would come either with some restrictions or significant programming overhead. The latter is even platform dependent and would therefore not meet the requirements. The Syntax of both, Javascript and Python, is clear and there are not many key words, which further underlines their simplicity. As all interpreted scripting languages, Javascript and Python are not as fast as lower level languages. For high performance applications, an interpreted language would certainly not be the first choice. However, performance had never been an issue for the given task as the timing constraints and the computational effort are very modest for a general purpose computer.

We have developed an application which provides a user interface and back-end for a wireless patient notification system. The application is platform independent and runs on almost any personal computer. The status of all sensor nodes in the wireless sensor network is monitored in the user interface in real-time. Whenever a patient calls a caretaker by pressing the button attached to a sensor node, an alarm message will be displayed in the GUI shortly after. As soon as the system operator confirms the alarm, the green LED on the sensor node will turn on to notify the patient that help is coming.

The system will soon be extended by an audio transmission feature to detect calls for help acoustically. This will enable patients who are not able to press a button to call a caretaker. Hopefully, the developed application will soon be used in a real world scenario and prove to be part of a useful and reliable patient notification system.

Appendix A

Adding new features

New features can easily be embedded into the existing code. As an example, let's assume we wanted to integrate a feature which displays the battery status of each node in the GUI. For simplicity, those packets shall be sent as separate messages once every minute. The steps that must be taken to integrate the new code include:

- define a new format for the packet payload, e.g. `"P,[node ID],[percentage]"`
- add an additional branch in the worker thread to handle these new packets
- pass the packets to the HTTP server by adding them to the input message queue
- add an additional branch in `getNodeStatus()` in the Javascript program (script.js) to handle this type of message
- display the battery status information in the GUI

Appendix B

User Guide

B.1 Files and folder structure

The table B.1 lists all important files and folders of the whole application and shortly describes their purpose.

B.2 How to use the application

- Make sure that Python 2.7 and Pyserial are installed on the computer.
- Connect the sink node and - if SMS notification is desired - a mobile phone via USB to the computer.
- Run the Python script, either with command line parameters or without. The syntax is as follows:

  ```
  main.py [server_port_number] [serial_port_number]
  ```

  The following output will be displayed:

  ```
  server started - localhost:80
  press ctrl+c to stop the server
  ```

- Open a web browser and connect to localhost.
- If no serial port has been passed to the server by the command line, the user will be asked to select the serial port to which the sink node is connected. The page shows all connected devices an refreshes automatically (see fig. B.1).
- The GUI will appear. It might take several seconds until the whole page is built up.

main.py	Python script
patnotify.db	Database (will be generated on first startup)
init.html	initialization page, will be displayed on first start-up to select the serial port
index.html	main page, to display the GUI
favicon.ico	icon for the main page
logfile.txt	log file for python script, generated dynamically
data	folder containing all necessary resource files (Javascript code, style sheet, icons, alarm sound)
data/jquery-1.8.3.min.js	jQuery library
data/script.js	Javascript code
data/style.css	style sheet for index.php

Table B.1: important files

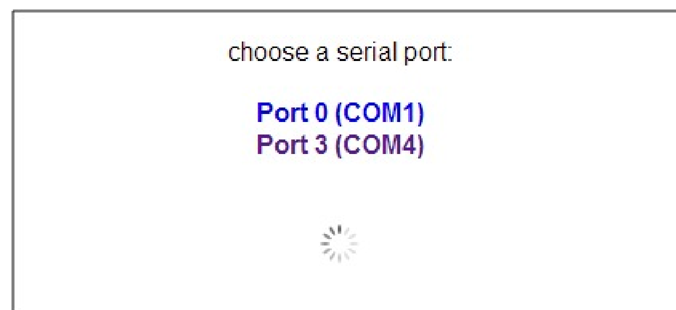

Figure B.1: initialization page

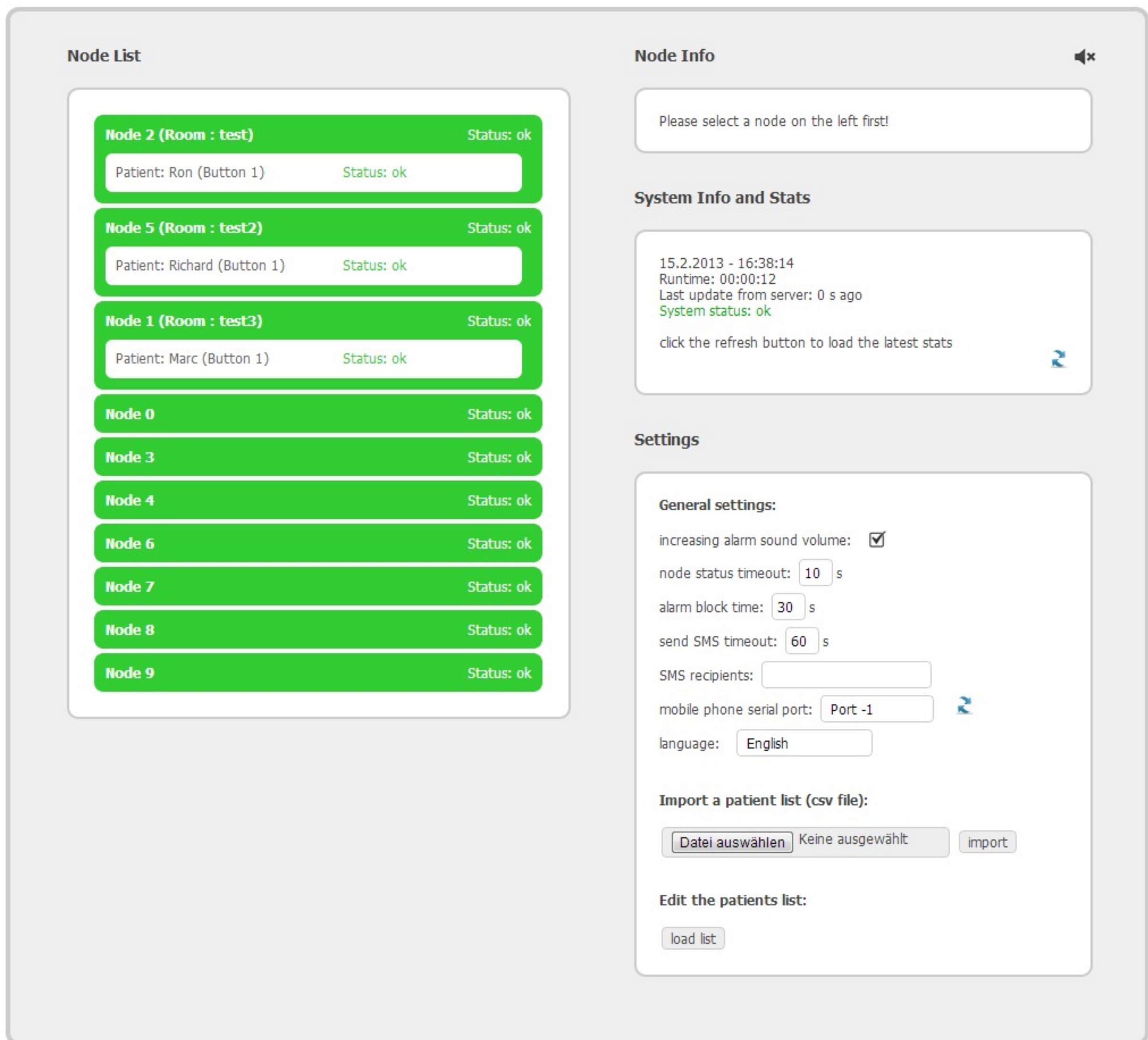

Figure B.2: screenshot of the GUI

B.3 Using the GUI

The user interface is divided into four parts: a list with all nodes (which is visible at all times), a node info box, a system status box and a settings box. All but the first field can be collapsed and expanded, depending on whether the user wants to see the field or not (see B.2).

- The node info box: When the operator selects a node by clicking onto it in the node list box, its status will be displayed in the node status field on the right. The status field includes information like the current state (OK or alarm) and how many times that this node has been lost and - if currently lost - for how long it has been lost.
- The system status box: This field displays information about the whole system, including date and time, runtime (elapsed time since last page reload), last message from the server, general system state and some statistics (e.g. how many alarms in the past hour). The main reason for this box is to display to the user that every component of the system is working properly. Whenever for whatever reason the server side Python script or the client side Javascript stops executing or the connection to the sink node is lost unexpectedly, it will be visible in this field.
- Settings Box: All configuration values which might be of interest can be changed directly from the GUI in the settings box. Furthermore, a CSV file containing a list of all patients can be imported. Those files can be created using an arbitrary spreadsheet application such as Excel.

	A	B	C	D
1	#Name	#Raum	#Node ID	#Button ID
2	patient1	roomA	0	1
3	patient2	roomA	0	0
4	patient3	roomB	1	0

Figure B.3: spreadsheet application

Lines starting with a number sign (#) are treated as comment and will be ignored. Opened as a text file, the content looks as follows:

```
#Name;#Raum;#Node ID;#Button ID
patient1;roomA;0;1
patient2;roomA;0;0
patient3;roomB;1;0
```

Whenever an alarm occurs, the border around the node list field will start to blink, the background of the pressed button will turn red and a confirmation button will appear (see 5.2). If the user confirms the alarm by clicking onto the homonymous button, a message will be sent back to the node which raised the alarm and the GUI goes back to normal display state (see B.2).

Note: If an alarm message is received from a button which is not in the list, an alert popup will be shown. Make sure every button which could possibly be pressed is in the list.

HOST_NAME	= "localhost"	
PORT_NUMBER	= 80	# default server port number
ACK_TIMEOUT	= 10	# if no ACK is received within 10s, the message is resent
ALARM_ACK_TIMEOUT	= 10	# if no ACK for an alarm is received within 10s, the alarm is resent to the GUI
BAUDRATE	= 115200	# baud rate for serial port communication
INIT_PAGE	= "init.html"	
MAIN_PAGE	= "index.html"	
DATABASE	= "patnotify.db"	
LOGFILE	= "logfile.txt"	
ALLOWED_CHARS	= "[^A-Za-z0-9 -]+"	# allowed characters for patient names, room, node and button ID
WAIT_TIMEOUT	= 8	# wait 8s for new status events before sending an empty 'alive' message to the client
USE_EMULATOR	= False	# use the emulator instead of reading from serial port
NODE_TIMEOUT	= 10	# the sink node must send data at least every 10 seconds
MAX_QUEUE_SIZE	= 20	# buffer size; keep this value small, otherwise the GUI will be flooded with messages when connecting to the server
FRAME_DELIMITER	= 0x7e	# framing delimiter, don't change this
ESCAPE_BYTE	= 0x7d	# encoding escape byte, don't change this

Table B.2: constants in the python script

B.4 Change the basic configuration

Usually, there will be no need to change the basic configuration and default values in the python script. However, it is allowed to change the global constants as well as the default values for the global variables in the file main.py. Most constants and variables should be self-explaining (see table B.2).

Note that the `WAIT_TIMEOUT` should be set to a smaller value than `NODE_TIMEOUT`. Otherwise, a "no data received from USB device" message will be sent to the client to warn the system operator that the sink node might have been accidentally disconnected or might not be working properly any more.

After the constants, there are some global variables. Besides the members of the class `Config()`, which holds the basic configuration and the default values, no other variables should be changed here. Note that the debugging level can be changed with the global variable `debugMode`. When enabled, all details are logged.

B.5 Configure SMS support

As already mentioned, there is a support for sending SMS to specified numbers for the case when an alarm stays unconfirmed for longer than a certain time (timeout). To use this functionality, a mobile phone must be connected to a USB port of the computer and a few settings in the GUI need to be configured:

- set the timeout for sending an SMS (default: 60 seconds)
- specify up to three mobile phone number (either with or without country code)
- select the correct serial port (refresh the list by clicking the icon next to the field)

Note that the connected mobile phone must be equipped with a valid SIM card with enough credit because the SMS will be sent over the cellular network of your carrier.

Appendix C

Program flow

C.1 Python application

```
Init:
  Get command line parameters
  Create log file
  Open the SQLite database, create all tables and default values if
       not already existing
  Read the configuration from DB
  Start a worker thread (which handles the serial port i/o)
  If serial port provided and available, connect to it
  Create an HTTP server (multithreaded, new thread for every new
      incoming request)
Serial Port I/O (worker thread):
  flush input and output buffer of the serial port
  create an empty list for all nodes and another list for all
      button events
  While not stopped
    Handle ack:
      Check if there are unconfirmed messages in the send list. If
          no acknowledgement for a message which was sent to the
          sink node has been received within [ACK_TIMEOUT] seconds,
           the message is resent
      Check every 10 seconds if an unconfirmed alarm has timed out,
           i.e. no confirmation has been received for an alarm
          within [ALARM_ACK_TIMEOUT]. If so, put the alarm message
          into the input message queue such that it will be
          retransmitted to the client and displayed on the GUI. If
          no confirmation for an alarm has been received within [
          sendSMSTimeout] seconds, an SMS is sent to one or more
          previously specified numbers. Therefore, a mobile phone
          needs to be connected to the computer and configured
          appropriately.
      Check if a new confirmation is available in the ack queue. If
           so, forward the confirmation to the sink node.
    Read data from serial port or get emulator data
    Analyse the received message
      If it is a status message:
        Check if the node already is in the list, if not, add it,
            otherwise update status
        Check if there is any node in the list for which no status
            msg has been received within the past [statustimeout]
            seconds; if so, set its status to zero
```

```
        If the status of one or more nodes has changed, write the
            whole list to the input message queue
      If it is an alarm message (button pressed):
        Check if there's already a pending alarm in the alarm event
             list, if not, add it, store the event in the database
            and append it to the input message queue, otherwise,
            ignore it
        Already confirmed alarms will be removed after a hold time
            of [alarmBlockTime] seconds.

HTTP server:
  respond to GET and POST requests
  on getstatus, check if there are any new messages in the input
      message queue and send it to the client; if no new status
      message has been received within 8 seconds, send the last
      known status

Cleanup:
  Wait for the worker thread to terminate
  Stop the HTTP server
  Close the database and log file
```

C.2 Javascript application

The program execution starts with the jQuery function `$(document).ready()`.

```
// check if browser supports HTML5 -> alert user if not
checkBrowser()
// load the language setting from the server
getLanguage()
// initialize the GUI
init()
  // load the text corresponding to the selected language
  loadLanguage()
  // create the fields/boxes, add titles and event handlers
  buildPage()
  // load the current settings from the server
  getSettings()
  // load the list of all patients from the server, create all
      necessary nodes, add all necessary   buttons and insert the
      patients names
  getPatientList()
  // load node status from server, update status of the nodes and
      add missing nodes to the list
  getNodeStatus()
  // display the system status field (auto refresh)
  updateSystemStatus()
  // refresh the node info box content
  updateNodeInfo()
```

There is a global list (array) of all nodes which represents the central data structure for the whole Javascript program. Its structure is as follows:

```
nodeList
[0] => { ID, status, buttons, statusinfo }
[1] => { ID, status, buttons, statusinfo }
[2] => ..
```

The element `buttons` itself is again an array including all buttons attached to a node:

```
buttons
[0] => { ID, name, room, status, counter }
[1] => { ID, name, room, status, counter }
[2] => ..
```

Most of the fields are self-explanatory. The last field of the array `nodeList` just holds some additional information about the node status (how many times a node has been lost and for how long). Similarly, the last field in the array buttons stores the number of times an alarm has been triggered.

Note that it is currently possible to assign a different room to each patient, i.e. two patients whose buttons are connected to the same node may be located in different rooms.

Appendix D

Debugging

Debugging was tricky at times. Whereas a C or Java compiler is quite strict and outputs warnings for every not explicitly stated type cast, scripting languages mostly don't bother about such things. This makes simple languages like Javascript or Python prone to errors. The programmer has to think about all possible cases. The developer tools integrated in the Internet Explorer showed to be very helpful for debugging the Javascript code. The documentation for Python explains all important methods and objects, but often lacks sample code or further explanations. As an example, the mime type and other properties have to be set appropriately in the header of the server response to an HTTP request, otherwise the browser won't recognize the returned files correctly. Another troublesome problem was the alarm sound, which somehow refused to loop. We were wrong in the assumption that the malfunction was caused by improper code on the client side. Instead, one single line in the header of the server response was missing and prevented the alarm sound from being looped. Such problems are rarely documented and finding a satisfying solution may be time consuming.

Bibliography

[1] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2012.

[2] Wireless sensor network. `http://en.wikipedia.org/wiki/Wireless_sensor_network`. Accessed: 2013-02-15.

[3] Cyclic redundancy check. `http://en.wikipedia.org/wiki/Crc16`. Accessed: 2013-02-15.

[4] Tos python implementation. `https://github.com/tinyos/tinyos-main/blob/master/support/sdk/python/tos.py`. Accessed: 2013-02-15.

[5] Tiny os: Serial communication. `http://www.tinyos.net/teps/doc/html/tep113.html`. Accessed: 2013-02-15.

[6] Point-to-point protocol. `http://en.wikipedia.org/wiki/Point-to-Point_Protocol`. Accessed: 2013-02-15.

[7] Serial line internet protocol. `http://en.wikipedia.org/wiki/Serial_Line_Internet_Protocol`. Accessed: 2013-02-15.

[8] High-level data link control. `http://en.wikipedia.org/wiki/High-Level_Data_Link_Control`. Accessed: 2013-02-15.

[9] pyserial v2.6 documentation. `http://pyserial.sourceforge.net`. Accessed: 2013-02-15.

[10] Xmlhttprequest. `http://en.wikipedia.org/wiki/XMLHttpRequest`. Accessed: 2013-02-15.

[11] jquery. `http://jquery.com`. Accessed: 2013-02-15.

[12] Sqlite. `http://www.sqlite.org`. Accessed: 2013-02-15.