University of Zurich[UZH]

ASSIGNMENT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Extending the Graphical User Interface CoMaDa with Contiki Support

Sebastian Pinegger
Zürich, Schweiz
Student ID: 10-933-802

Supervisor: Dr. Corinna Schmitt, Lisa Kristiana
Date of Submission: September 30, 2015

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Assignment
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

In this thesis, a new graphical user interface (GUI) for support of Contiki is presented. The GUI is needed to configure sensors with Contiki. The implementation is robust and will be used to work with the Contiki platform. The integration extends the already existing tool CoMaDa. The implementation is written for Contiki 2.7 and integrates the TinyIPFIX protocol.

Zusammenfassung

In der vorliegenden Arbeit, wird das neue Graphical User Interface (GUI) mit Contiki Unterstützung vorgestellt. Das GUI wird benötigt um Sensoren, welche auf Contiki laufen zu untersützen. Die Umsetzung ist robust und wird mit der Contiki Plattform funktionieren. Die Funktionalität erweitert die bereits bestehende CoMaDa Applikation. Die Umsetzung ist für Contiki 2.7 geschrieben worden und integriert das TinyIPFIX Protokoll.

Acknowledgments

First and foremost, I want to thank my supervisor Dr. Corinna Schmitt for her continuous feedback and support. Additionally, I want to thank Professor Burkhard Stiller, head of the Communication Systems Group at the University of Zurich, for providing me with the chance to conduct the assignment at his research group. Third I want to thank Michael Meister for his support and feedback during the implementation. Last but not least I want to thank Claudio Anliker for the conversations over lunch and providing me with inputs and ideas for the implementation. Specially I want to mention the ongoing support of my parents, Barbara and Dr. Thomas Pinegger, without them my studies at the University of Zurich would not be possible. I am especially grateful for my friends, Patrick Tanner, Mathias Roth and Nicolas Lanz, to support and motivate me in times of doubt.

Contents

Chapter 1

Introduction

1.1 Motivation

Due to the growth of the Internet and the device diversity together with their communication capability, the Internet-Of-Things (IoT) determines a highly relevant topic as of today. IoT is not limited to Client-Server(C/S) architectures, Peer-to-Peer (P2P) networks, and well-known devices like server, computer, and routers any more. It especially includes wireless sensor devices connected within a Wireless Sensor Network (WSN) [4], [5].

The application range of those WSNs reaches from intelligent homes via logistics and health care to environmental monitoring. Together with the mobility of end users, the request to stay informed, and the concept of IoT new opportunities such as to observe and manage deployed (wireless sensor) networks using mobile devices (e.g., Smartphones or handhelds) have emerged quickly [5]. Many application-driven solutions exist, but are not generalized and are not fully hardware-independent. The global request by users is a Graphical User Interface (GUI) allowing a user-friendly interaction with WSN-networked devices and a easy-to-use visualization of the deployed network and its data, independent pf data types, hardware, and network size. The first solution addressing those requirements was developed as CoMaDa [3], a framework for configuration, management, and data handling of WSNs, working with a visualization in real-time of the deployed WSN. An extension and improvement for remote access to such WSN configuration and management functionality was reached with WebMaDa [6]. In the beginning of this assignment, the CoMaDa only supported TinyOS. In the previous years Contiki has gained a lot attention in the community.

1.2 Description of Work

The work of this assignment is threefold. First, the existing version of CoMaDa (from now on called WSNFramework) had to be analyzed, documented in case of faults, and currently

existing error, e.g., online visulization of received data Xively, had to be corrected to reach a very stable prototype.

Within the second step the existing programming functionality of the devices with TinyOS had to be extended with Contiki [1] support following the existing design of TinyOS (e.g., Basestation and node programming or tunnel activation).

In the third and last step the visualization of the network topology and received data has to be specified and prototyped into a running version, including storage specifications for upcoming implementations.

1.3 Report Outline

The report for the assignment is conceptually structured as follows: First, the related works are highlighted in Chapter 2 to frame this report into a context. Following up on the related works in Chapter 3.1.1, the focus lays on the analysis of the WSNFramework. This part is essential to understand the thoughts on the implementation. It is split into two parts, the client and the server analysis. Then the implemented work is highlighted in Chapter 3.1.2, with a particular focus on the implementation of the GUI for Contiki. This includes a deeper insight on how the implementation decisions had been made. In Chapter 3.1.3 the report goes into more detail of the documentation for the implementation. A central part of this report a is the architectural outline in Chapter 3.2. This part is threefold, as it shows the architecture of the client and server, and additionally gives examples how the client and server communicate. The report ends with the Chapters 4 and 5, in which the evaluation of the analysis and implementation is discussed and to round up the report, it is finished with a summary and conclusion.

Chapter 2

Related Work

The CoMaDa application was written byAndre Freitag. The idea was to have an framework which could handle the configuration of TinyOS nodes and to connect a Wireless Sensor Network with the cloud. From his Bachelor's Thesis evolves the WSNFramework, which was enriched with more funcitonalities by other students. The WSNFramework consists of two part, a GUI to interact with the user and the server holding the functionality, including the construction of the website, communicating with the nodes, and the upload to the cloud.

Michael Meister is currently working on the TinyIPFIX protocol for Contiki based nodes. A part of his work was integrated in this assignment. The functionality includes the conversion of data gathered by the nodes to the CoMaDa application.

Chapter 3

Implementation & Architecture

This chapter describes how the extension for Contiki Support was implemented for this assignment. Thus, the basic framework of the WSNFramework is outlined, followed by the software architecture of the newly integrated GUI and server-side logic. The resulting implementation includes the full communication between the server and client. Additionally it is indicated where to change the configurations for the Contiki Support.

3.1 Implementation

This section highlights the steps, which are outlined in Section 1.2 and implemented. Starting with the analysis, the already existing WSNFramework was intensively examined. The main part is the implementation, while the technical aspects are discussed in the section Section 3.2. In the third part the documentation for the visualization of the gathered data by the sensor network is described.

3.1.1 Analysis

The WSNFramework is very complex and integrates the WSN of TinyOS very well. On first sight it seems well structured and documented in detail. But at a closer look some problems arise. The WSN is split into two parts, server side and client side. The server is written in Java and uses standard libraries. The client side consists of HTML, CSS and Javascript. On some occasions PHP is used to support dynamic construction of the HTML code on the server side. To give a better overview let us have a closer look at the file structure of the WSNFramework, and then get into detail about the framework.

Analysis Server

The server part of the WSNFramework is written in Java (Version 1.7.0_79) and uses standard libraries. It handles the logic of the WSNFramework; this includes the communication with the command line tool, construction of the web interface, and the event

handler for the nodes. The server creates an application called WSNApp app, in which all the modules are attached. As you can see in Figure 3.1, the source code consists of multiple modules.

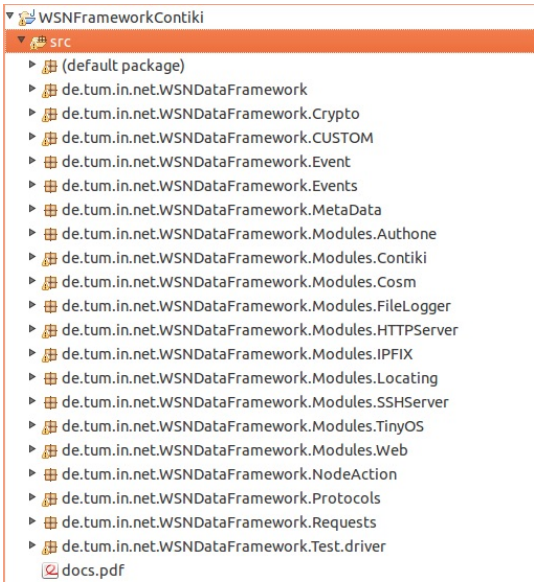

Figure 3.1: File Structure of Server

Each module holds a main class, which is attached to the WSNApp app with app.addModule (e.g., app.addModule(new TinyOSHelperModule("/opt/tinyos-2.1.2")), which is part of the module, shown in Figure 3.2). During this process it registers all the functions and adds the URLs to the HTTP module of the WSNApp app. In this case localhost:8000/tiny would be controlled by the HTTPController in the module shown in figure 3.2. Please refer to Section 3.2.3 for a more detailed plan on how the mapping between an URL and the corresponding function works.

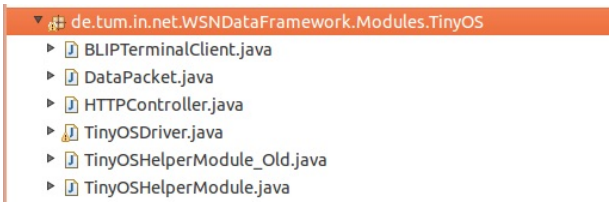

Figure 3.2: File Structure of a Module

The already existing server implementation contains commented out code in multiple places. In Figure 3.2 for example are two classes beginning with "TinyOSHelperModule", but one ends with _Old. The naming convention of package is not consistent throughout the WSNFramework, e.g. *de.tum.in.net.WSNDataFramework.Modules.TinyOS* contains capital letters, which should be avoided [2]. This raises some questions, but refactoring is not in scope of this work. It should be tackled in the future.

Another open problem is that often the constructor of a class is marked with "@SuppressWarnings("unchecked")", as it contains Java bytecode of another class. It may also have

other roots, but this needs time to fix and might be solved in newer Java Versions. Fixing the problem with the compiler would go beyond the constraints of this work.

Another interesting aspect of the framework is the HTTPServer module and the handling of URLs, which was mentioned before. Some parts of the URL are hard-coded and some are dynamically constructed, when the WSNApp app is being started. The outlined factors hinder to easily understand the framework and its componend.

Analysis Client

The front-end is written in HTML, CSS, and JavaScript. As already existing framework JQuery is chosen. For the communication AJAX is used. Instead of XML the frame work uses JSON as the data-interchange format.

All the files for the website are stored in the folder ”html”, as you would expect from a web server. In Figure 3.2, we see the file structure of the website. Every sub domain should be stored in a sub folder [7]. In the structure of the WSNFramework we see a partial implementation of the methodology. The sub folder exist for certain web pages, but in the folder ”index” are the files dynamic.html, nodes.html, etc. stored. The paradigm to separate web pages is broken and makes navigation more challenging, if you are not familiar with the WSNFramework.

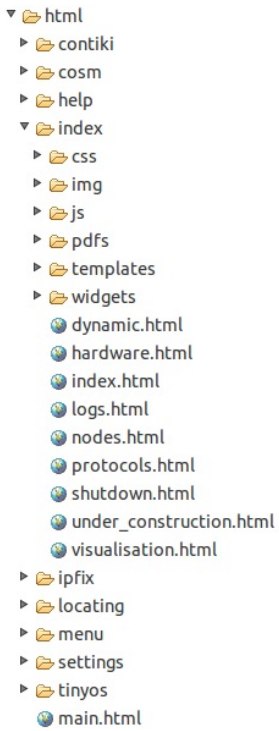

Figure 3.3: File Structure of Client

Seen in Figure 3.3, the folder ”index” holds a folder ”js” and ”css”. Based on the methodology by www.thehelloworldprogram.com [7], they should be stored in the top folder. The difficulty is to traverse from a sub folder in ”html” to another sub folder. For example, if

a file in ”contiki” wants to access a Javascript file in ”js” it has to go up one folder and then down again.

Leaving the structural aspect, the focus is being put to the code itself. The paradigm for web development indicates, that the three components for a web page, HTML, CSS and Javascript, are decoupled [7]. It makes it easier to read and also separates functions, style elements and markups. Another advantage is to have common ”best practise”, that gives an outsider the possibility to easily adopt new functionality and style.

In the folder ”html” exists a file called ”main.html”. It holds some global content, like the navigation bar, a calculator to convert hexadecimal, decimal and binary numbers, and some functions for the TinyOS integration. Every web page loads the main.html and inserts the corresponding content into the body. It is a common practice, but the ”main.html” file breaks the decoupling paradigm outlined in the paragraph before. An example is shown in the Figure 3.4.

In Figure 3.4 the lines 154 - 160 hold a Javascript function, which could stored in a file within the folder ”js”. The line 168 hold some information of the style of the box. Normally all styles are stored in a CSS file to make the markup more readable. Followed by the line 169 we encounter the same occurrence.

```
154             var width = 0;
155             $('ul.sf-menu').children().each(function(){
156                 width += $(this).width()+1;
157             });
158             $('#menu-wrapper').width(width);
159         });
160         </script>
161     </div>
162
163     <div id="body">
164     </div>
165
166     <div id="tools">
167         <!-- Tools -->
168         <div style="position: fixed; right: 6px; bottom: 13px; height: 13px; width: 143px; padding: 3px 13px 7px 13px; background-color: #455875;">
169             <a href="#" style="color: white; font-weight: bold;" onclick="return toggle_numberConverter();">Number-Converter</a>
170         </div>
```

Figure 3.4: Example of Decoupling

3.1.2 Implementation GUI & Server

The implementation consists of two parts. The GUI is outlined in the beginning of this subsection and the server in the last paragraph.

The GUI for Contiki slightly differs from the GUI of TinyOS. It does contain the same functionality including the configuration of nodes and border router, and the tunnel activation. The configuration has the ability to set a global parameter for the right platform (e.g., sky, esp, etc.). The user is able to choose between a node configuration, a border router configuration or the tunnel activation. It then differs between node and border router as seen in Figure 3.5.

During a configuration of a node or border router the user is able to choose the platform of the node or border router (Figure: 3.6). At the same time the user is able to choose from a predefined project. Please refer to Section 3.2 for more details concerning the configuration of predefined projects on nodes and border routers.

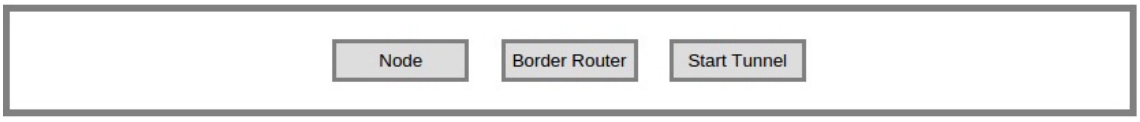

Figure 3.5: web page of Contiki GUI

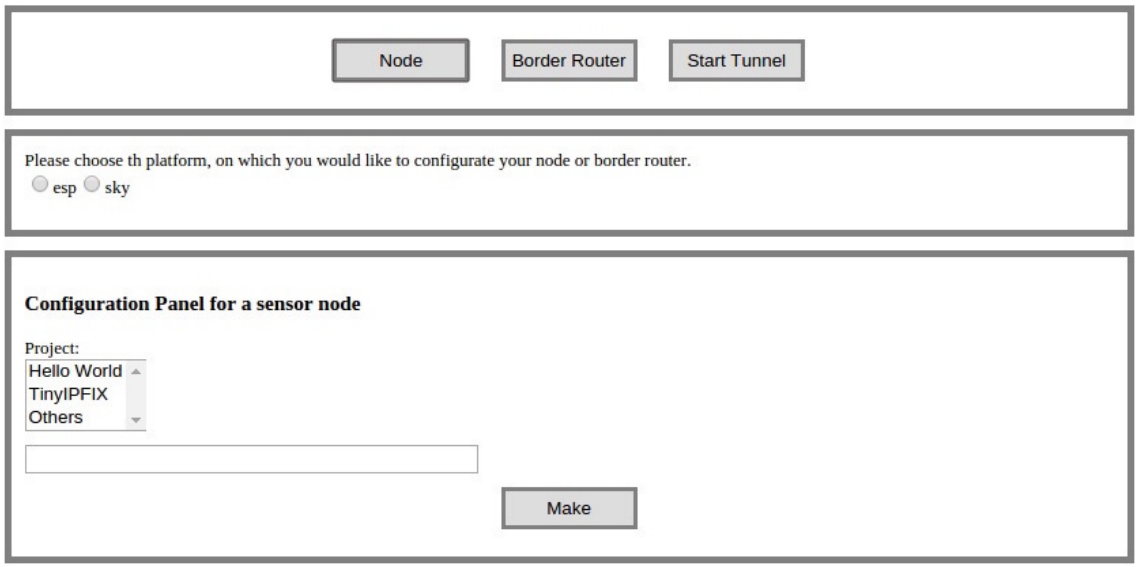

Figure 3.6: Configuration Panel for Contiki Node

Additionally to the predefined projects, the user is able to define an own projects. This can be added, when from the dropdown ”Others” has been chosen. The content on the right hand side dynamically loaded, when a project is chosen from the left hand side (Figure: 3.7). For security reasons it is not possible to directly get a path from the browser [8]. The path has to be an absolute path starting with /home, when the WSNFramework is deployed on a linux machine.

In Figure 3.8 the GUI is working on a request, after the ”Make” button has been pressed. The ”Make” button triggers an AJAX call to the server. When the server runs the commands, the GUI displays a loading sign and disable the ”Make” button. Additionally it swaps the content of the configuration box with a messages, that it is working on the make command and displays a reset button for the GUI. This makes the GUI very dynamic and gives the user a feedback that the server is working on the request.

After the make request, the client receives data for the shell and the information of the usage of RAM and ROM. The shell box on the bottom is updated with the content of the reply from the server. The box has a fixed size, but within it is scrollable. The information on RAM and ROM is loaded into the GUI as soon it has been received by the logic of the client. The install button next to the reset button is activated right after the construction of the RAM and ROM information in the HTML file (Figure 3.9).

The configuration flow of a node sensor and a border router are exactly the same. The display of RAM and ROM is always based on the last make command and does not change, when you switch between node and border router.

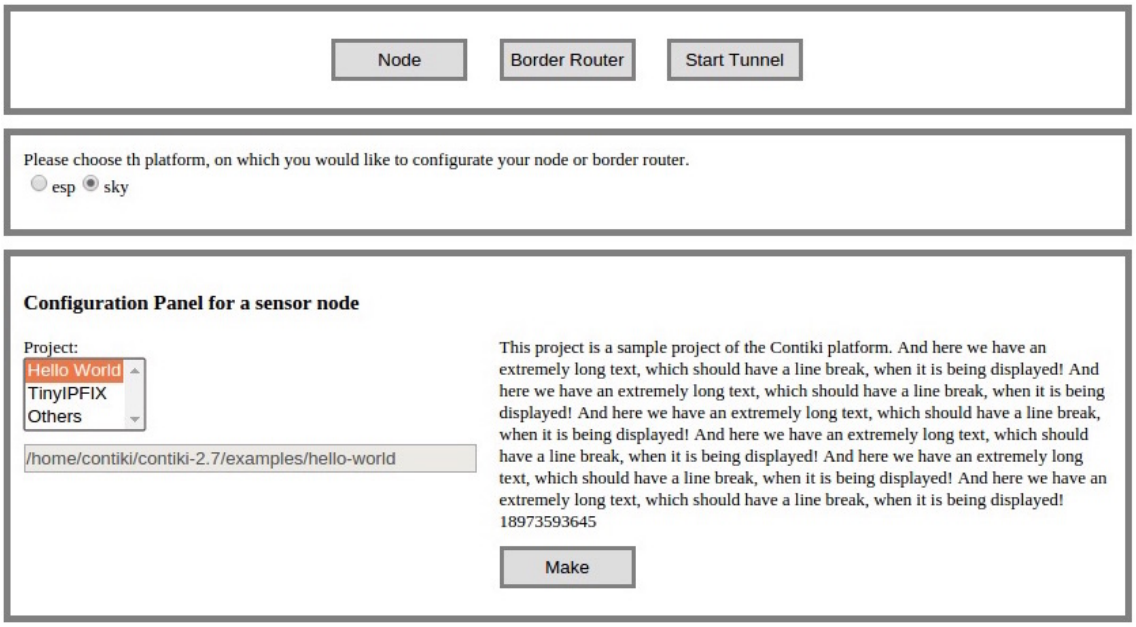

Figure 3.7: Dynamic Configuration Panel for Contiki Node

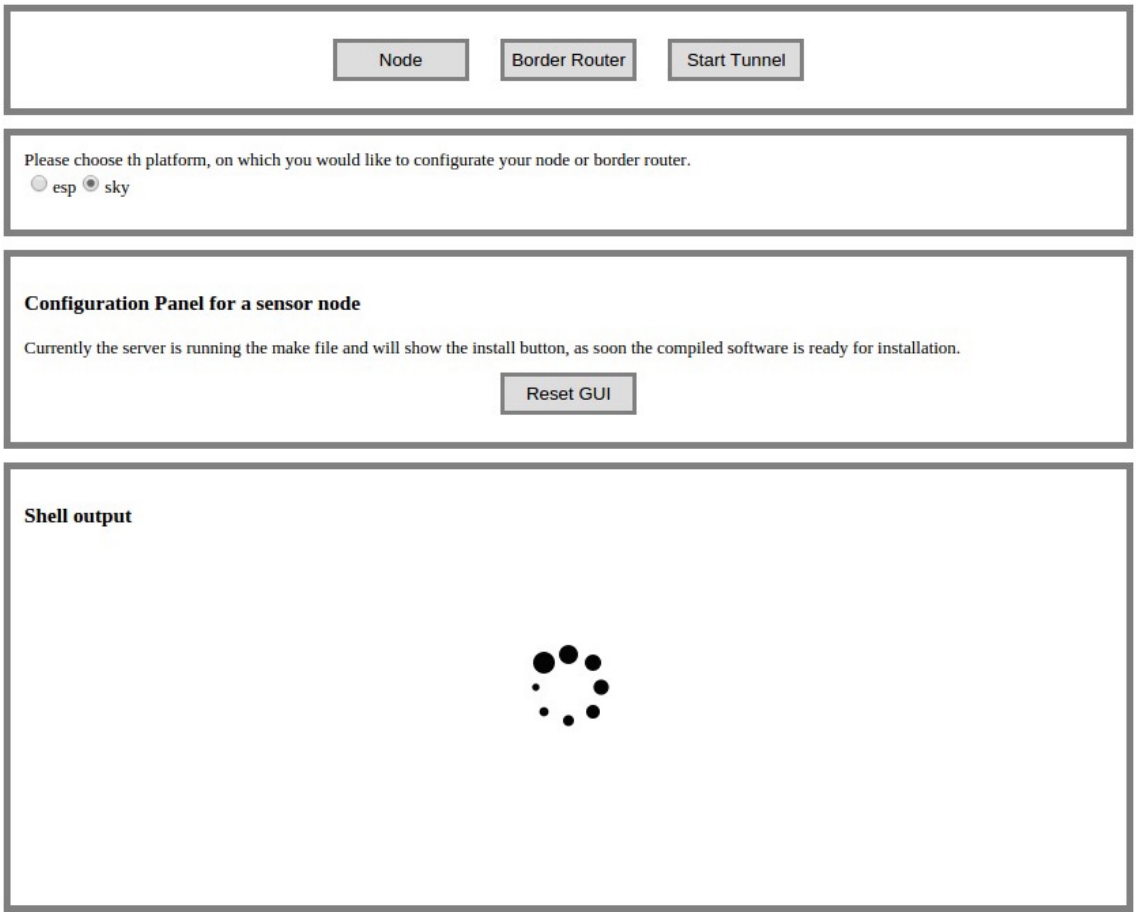

Figure 3.8: Waiting Screen for Contiki Configuration

Figure 3.9: Install screen for a Contiki Node/Border Router

The server handles all the calls from the GUI to fulfill the make and install triggered by the GUI. The implementation for the server logic constis of two parts. The class *HTTP-Controller.java* manages all the HTTP Requests and prepares the parameters for function calls. The class *WSNContikiModule.java* gets the parameters from the HTTPController and executes shell commands. It contains the business logic as well. For a more technical overview please refer to Section 3.2.

3.1.3 Documentation

During the initialization of the GUI call the server to get the predefined projects and platforms into the front-end. All of them are defined on the server-side and send within a JSON file to the client. The parameters are defined in the class *WSNContikiModule.java*. This function on the other hand is called by the *HTTPController.java* .

WSNContikiModule.java Location:*/home/contiki/workspace/WSNFrameworkContiki /src/de/tum/in/net/WSNDataFramework/Modules/Contiki/WSNContikiModule.java*

Listing 3.1: Configuration of projects

```
/**
 * In this method are all the project names, working directions and
    descriptions stored.
 * It is called by HTTPController, when the GUI request all the configuration
    parameters.
 * */
public JSONObject getConfigs(){
   //The first argument is the name of the project, the second one is the
      working directory, the third one includes a small summary of the project
   //Add node projects
   JSONObject helloWorld = new JSONObject();
   helloWorld.put("projectName", "Hello World");
   helloWorld.put("workingDr",
      "/home/contiki/contiki-2.7/examples/hello-world");
   helloWorld.put("description", "This project is a sample project of the
      Contiki platform.");

   JSONObject node = new JSONObject();
   node.put("projectName", "TinyIPFIX");
   node.put("workingDr", "/home/contiki/contiki-projects/contiki-node");
   node.put("description", "This module is the TinyIPFIX Protocol for
      Contiki.");

   //Merge node configurations
   JSONArray nodes = new JSONArray();
   nodes.add(helloWorld);
   nodes.add(node);

   //Add border projects
   JSONObject borderNode = new JSONObject();
   borderNode.put("projectName", "Border Router");
   borderNode.put("workingDr",
      "/home/contiki/contiki-2.7/examples/ipv6/rpl-border-router/");
   borderNode.put("description", "This configuration is the basic border
      router configuration.");

   //Merge border configurations
   JSONArray borders = new JSONArray();
   borders.add(borderNode);

   //Add platforms
   JSONObject sky = new JSONObject();
   sky.put("platformName", "sky");

   JSONObject esp = new JSONObject();
   esp.put("platformName", "esp");
```

```
39
40    //Merge platforms
41    JSONArray platforms = new JSONArray();
42    platforms.add(esp);
43    platforms.add(sky);
44
45    //Merge nodes configs and border router configs
46    JSONObject configs = new JSONObject();
47    configs.put("nodes", nodes);
48    configs.put("borders", borders);
49    configs.put("platforms", platforms);
50
51    return configs;
52 }
```

In Listing 3.1, the configuration of the pre-defined projects is outlined. To add a node projects or a border router projects, another *JSONObject* has to be added either on the *JSONArray* ”nodes” or ”borders”. A *JSONObject* for a node or border router consists of three parts, the parameter IDs are *projectName*, *workinDr*, or *description*. The platform parameter are set up in the same manner, multiple *JSONObject* in a *JSONArray*. The only difference is here, that a *JSONObject* only has one entry. The *JSONObject* has the ID *platformName*.

3.2 Architecture

In the section before only the conceptional work on the GUI and partially the technical implementation of the server logic has been shown. This section digs deeper into the technical aspects of the client and the server. In the third subsection an example of the communication between the server and the client is shown.

3.2.1 Server

The existing file structure was not changed.The implementation of the server part is exclusively done in the package *de.tum.in.net.WSNDataFramework.Modules.Contiki*. In Figure 3.10 both class *HTTPController.java* and *WSNContikiModule.java* are listed.

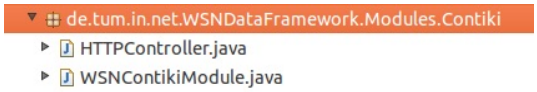

Figure 3.10: File Structure of the Contiki Module

HTTPController.java: This file holds all the functions for the AJAX calls from the client. To better understand how it works, some example code is shown in Listing 3.2. During a AJAX call the *HTTPServer.java* parses the URL of the call. The first sub path

indicates the controller, for example: http://localhost:8000/contiki would call this controller. The second sub path indicates the action. http://localhost:8000/contiki/domake resolves into the action "domake", the HTTPServer then calls the "domakeAction" function within the *HTTPController.java*. An example is given in the listing bellow. The file is located in */home/contiki/workspace/WSNFrameworkContiki/src/de/tum/in/net/-WSNDataFramework/Modules/Contiki*.

HTTPController.java, it shows the first part of the domakeAction function

Listing 3.2: domakeAction() in HTTPController.java

```
/**
 * @param: json file with command configurations
 * @return: json file with the output of the shell and information on RAM and
    ROM
 *
 * This function calls the domake function in WSNContikiModule
 * to execute the shell command, which is also constructed in it
 * */
public void domakeAction(HTTPRequest request, HTTPResponse response){
   Map<String,String> jsonResult = new LinkedHashMap<String,String>();
   ArrayList<String> outputMake = new ArrayList<String>();
   ArrayList<String> outputRamRom = new ArrayList<String>();

   //Read parameters from json file
   String platform = request.arguments.get("platform").toString();
   String cmdClean = "make clean";
   String cmd = "make TARGET=" + platform;
   String workingDr = request.arguments.get("workingDr").toString();

   //Start Commands
   WSNContikiModule module = (WSNContikiModule)this.module();
   module.callShell(cmdClean, workingDr);
   outputMake = module.callShell(cmd, workingDr);
```

WSNContikiModule.java: This class is attached to the WSNApp app during the initialization of the WSNFramework. It registers the HTTPController and executes all the shell commands. It also holds the configuration parameters as mentioned before. The file is located in */home/contiki/workspace/WSNFrameworkContiki/src/de/-tum/in/net/WSNDataFramework/Modules/Contiki*.

WSNContikiModule.java, the code bellow is the first part of the command which is triggered by code example above.

Listing 3.3: Function callShell(...) in WSNContikiModule.java

```
/**
 * This Method is used to communicate with the shell
 * */
public ArrayList<String> callShell(String cmd, String workingDr){
   ArrayList<String> output = new ArrayList<String>();
```

```
6     ArrayList<String> fullCommand = new ArrayList<String>();
7
8     //Construct the commands in an array for
9     fullCommand = commandConstructor(cmd);
10
11    //Shell call and read
12    try {
13       Process p = null;
14       ProcessBuilder pb = new ProcessBuilder(fullCommand.toArray(new
             String[fullCommand.size()]));
15       pb.directory(new File(workingDr));
16       p = pb.start();
```

One major problem with running TinyOS and Contiki at the same time is, that the tool chains are not compatible with each other. This causes compatibility problems within in the WSNFramework. During the initialization of WSNApp app multiple TinyOS drivers are added to the instant. Therefore, when the WSNFramework is configured for Contiki it can not access the TinyOS tool chain. Ultimately it causes the GUI to show a message that the TinyOS drivers are not available, as they have to be commented out in the WSNFramework.

3.2.2 Client

The GUI was constructed with the paradigms outlined in the Section 3.1.1 in mind. For simplify matters, no PHP is used in the web page of the Contiki GUI. To ensure an easy and understandable setup, only Javascript, CSS, and HTML are being used for the integration of Contiki Support. The use of the technologies are explained in more detail. In Figure 3.11 are the three files index.html, contiki.js, and contiki.css shown. Those three will be explained in the following paragraphs.

index.html: This file holds all the elements for the GUI. It can be seen as a container with all the elements, which are then constructed with the help of the javascript functions in contiki.js. The file is located in */home/contiki/workspace/WSNFrameworkContiki/html/-contiki*.

contiki.js: This file holds all the logic to assemble the GUI, making the AJAX calls and as well as showing the output of the shell of the local server. The file is located in */home/contiki/workspace/WSNFrameworkContiki/html/index/js*.

contiki.css: This file holds all the style sheets for the Contiki GUI. The file is located in */home/contiki/workspace/WSNFrameworkContiki/html/index/css*.

In the Listing below we have a closer look at the function **makeNode()**. This function handles the communication with the server, after the ”make” command has given by the users. In the first two lines it extracts the chosen project (or when defined, the path for the project) and the chosen platform. Followed by 7 lines it rearranges the GUI. The shell box at the bottom of the GUI is displayed with a loading sign. The ”make” button and the

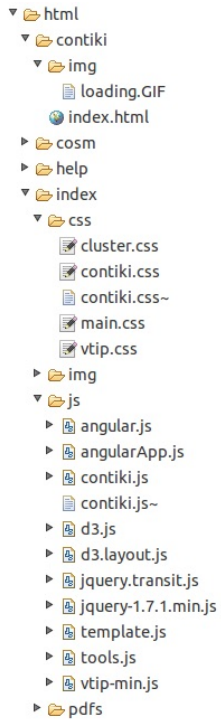

Figure 3.11: File Structure of the Contiki GUI

configuration of the project are hidden. The reset button will be shown and the content of the configuration box is loaded with information of the process. After the reconstruction of the GUI, the AJAX call is triggered. If the call was successful the GUI is updated, this includes the information of RAM and ROM, the shell output in the shell box, and the ”install” button is shown.

Listing 3.4: Function to communicate with the server, when called by the make button

```
1  //This function triggers the make of the node.
2  function makeNode(){
3    var workingDr = document.getElementById('makeNodeFilePath').value;
4    var platform = $('input[name=platform]:checked', '#platform_selection').val();
5    loading();
6    $('#makeNodeButton').hide();
7    $('#node_configuration_content_and_information').hide();
8    $('#resetNodeButton').show();
9    $('#node_information').show();
10   var informationNode = "Currently the server is running the make file and will show
         the install button, as soon the compiled software is ready for installation.";
11   document.getElementById('node_information').innerHTML = informationNode;
12   $.ajax({
13     "url": "/contiki/domake",
14     data: {"workingDr": workingDr,
15     "platform": platform},
16     context: this,
17     dataType: "json",
18     success: function(response){
19       displayRAMROM(response);
20       shellOutput(response);
21       $('#installNodeButton').show();
```

```
22      }
23    });
24  }
```

3.2.3 Communication: Server - Client

In conclusion, in this Section, the spotlight is set on an example of the communication between those two. In the Figure 3.12 the reader can find the workflow of a make-command from the GUI. On the left hand side the reader sees the client, it is controller by the user. On the right hand side the server is shown.

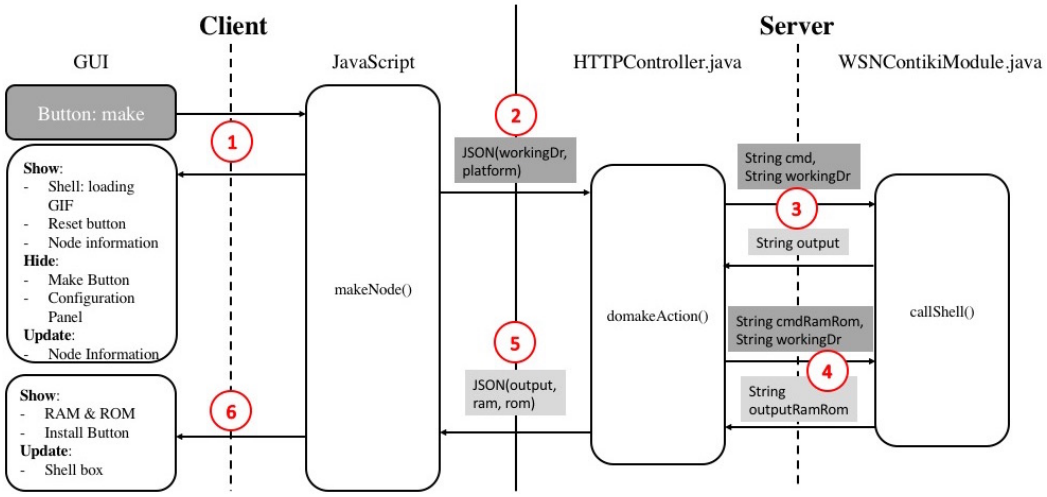

Figure 3.12: Communication between Client and Server

In step 1, after the user pressed the ”make” button, the JavaScript on the client side updates the GUI as follows. It inserts a loading GIF into the shell as seen in figure 3.8, the ”reset” button, and displays a node, that the server is working on the request. At the same time the ”make” button and the configuration panel are hidden. In step 2 the JavaScript makes the AJAX call to */contiki/domake* with the two parameters ”workingDr” and ”platform” in a JSON file. On the server side, the *domakeAction(...)* function calls the *callShell(...)* twice. During the first time, indicated in step 3, a make-command is run to make the make-file in the project folder. The output is locally stored in *domakeAction(...)*. In the second call, step 4, the *domakeAction(...)* function extracts the information on RAM and ROM form the compiled code. (Step 5) As soon the make-command and extraction of RAM and ROM has run, the *domakeAction(...)* function returns a JSON-file with the information of the shell output of the make-command, RAM, and ROM. On the client side, the AJAX call, waits for a response and when it receives the return it updates the GUI. (Step 6) As seen in Figure 3.9, a box with the RAM and ROM information is added. The install button is made available and the shell output is updated according to the content of the JSON file.

Chapter 4

Evaluation

The WSNFramework offers a lot of functionality already for TinyOS. With the integration for Contiki Support it gains extended functionality, but comes to its limits of the framework. Starting with the analysis, it became clear in a early stage, that the integration of Contiki Support will cause hick ups. The java code on the server is well designed, but breaks multiple paradigms of the naming conventions. The same is laid out in this assignment for the file structure of the client.

The implementing phase for this assignment showed the construction of the GUI and the server-side logic to enrich the GUI with the required functionality. To highlight here, the analysis was not sufficient enough to have a smooth implementation. On multiple occasions, debugging was hassle and thanks to the help of other students, who worked on the same framework, the obstacles could be solved. To check the functionality of the GUI, multiple nodes were configured with it. In the field test they worked, as expected.

The documentation phase was rather quickly done, as it did not require new implementations. The predefined projects, as mentioned in the Section, build the core of the third implementation phase of this assignment.

Chapter 5

Summary and Conclusions

To sum up this assignment I want to quickly outline the steps of the work, that have been done. The analysis included debugging the components and simulate the work flows with TinyOS to project the same logic on the Contiki Support. During the implementation phase of the Contiki Support, it became clear that this can not be done as outlined in the analysis phase. Many challenges arose and the framework had to be analyzed in more detail with a special focus on the modules within the framework and their interactions with each other. The rough design of the GUI was done first with an intuitive work flow. The GUI and the server were implemented simultaneously. Each trigger of logic on the GUI had to have a back-end function to handle the request. During this phase the predefined projects should not have been hard coded in the java class. Finishing up the documentation with the configuration documentation and the final report, laying in front of you, the assignment came to an end.

The assignment was more challenging than thought in the beginning. The server is well structured in terms of file management, but the source code itself is challenging to understand. It did cost a lot of time in the beginning, during the analysis, to get used to the framework. The implementation of the Java code was done in a timely manner. On the client side, the file structure could be improved and a refactoring and rearranging of files and code is highly recommended. In general the whole framework should be a standalone application for Contiki Support only and therefore being freed from all TinyOS components. This integration of the tunnel could not be accomplished. This feature is missing and needs to be implemented to reach the same functionality as the GUI of TinyOS.

Bibliography

[1] Contiki: The Open Source OS for the Internet of Things. `http://www.contiki-os.org/`, last visit: 29.09.2015.

[2] Naming a Package. `https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html`, last visit: 29.09.2015.

[3] André Freitag, Corinna Schmitt, Georg Carle. *CoMaDa: An Adaptive Framework with Graphical Support for Configuration; 9th International Conference on Network and Service Management*. Zurich, Switzerland, October 2013.

[4] John Paul Walters, Zhengqiang Liang, Weisong Shi and Vipin Chaudhary. Wireless sensor network security: A survey. *Security in Distributed, Gris, and Pervasive Computing, Auerbach Publications, CRC Press*, 2006.

[5] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley-Interscience, 2007.

[6] Michael Keller. Design and Implementation of a Mobile App to Access and Manage Wireless Sensor Networks. `http:/files.ifi.uzh.ch/CSG/staff/schmitt/Extern/Theses/Michael_Keller_MA.pdf`, last visit: 29.09.2015. Bachelor Thesis, University Zurich, Communication Systems Group, Department of Informatics.

[7] JR Nielsen. Organizing Files and Folder Structure for Web Pages. `http://www.thehelloworldprogram.com/web-development/creating-files-folder-structure-web-pages/`, last visit: 29.09.2015, June 2014.

[8] User: Vohuman. StackOverflow. `https://stackoverflow.com/questions/15201071/how-to-get-full-path-of-selected-file-on-change-of-input-type-file-usi`, last visit: 29.09.2015, March 2013.

Abbreviations

GUI	Graphical User Interface
WSN	Wireless Sensor Network
CSS	Cascading Style Sheets
JS, js	JavaScript
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
AJAX	asynchronous JavaScript and XML
XML	Extensible Markup Language
CoMaDa	**Co**nfiguration, **Ma**nagement and **Da**ta handling Framework

Glossary

CoMaDa CoMaDa is a framework for configuring nodes and acting as a gateway between a Wireless Sensor Network and the Internet.

WSNFramework The WSNFramework is the technical implementation of CoMaDa.

website A website is made up of a number of different web pages connected by links.

web page A web page is one single page of information.

TinyOS TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters.

Contiki Contiki is an open source operating system for the Internet of Things. Contiki connects tiny low-cost, low-power microcontrollers to the Internet.

List of Figures