CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Testbed for Internet of Things and Wireless Sensor Networks

Master's Thesis in Computer Science and Engineering

FAHAD LAFTA
COLEB MUJURIZI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

MASTER'S THESIS 2016:NN

Testbed for Internet of Things and Wireless Sensor Networks

FAHAD LAFTA
COLEB MUJURIZI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Testbed for Internet of Things and Wireless Sensor Networks

FAHAD LAFTA
COLEB MUJURIZI

Supervisors:
Olaf Landsiedel, Department of Computer Science and Engineering
Beshr Al-Nahas, Department of Computer Science and Engineering

Examiner:
Magnus Almgren, Department of Computer Science and Engineering

Master's Thesis 2016:NN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

Time is ripe for what has been envisioned in the past two decades - the possibility of interconnecting everyday things to collect and exchange data amongst themselves, popularly known as the Internet of Things (IoT). The Internet of Things is the newest communication paradigm which is fuelling productivity and efficiency, and simplifying lives of people in different areas such as home automation where connected devices can handle more operations such as regulating the temperature, controlling the lighting, and monitoring the home hence freeing up the home owner to concentrate on other tasks. Self-driving cars which heavily depend on the IoT architecture are being introduced in the automotive industry to reduce road accidents and increase efficiency of road users. Other industries are also embracing IoT to boost their efficiency. IoT devices are equipped with processors to perform computations on the data, actuators and sensors to interact with the environment, and IP addresses to communicate over the Internet. In an IoT ecosystem exists a Wireless Sensor Network (WSN), which is a network of low powered electronic nodes that consist of different sensors which are used in collecting data from the environment. Achieving a seamlessly integrated IoT and WSN architecture is complex because of the heterogeneous nature of the devices, link layer technologies, and the services involved. By "heterogeneous", we mean the use of different hardware architectures. It is harder for developers of IoT applications and WSN protocols to carry out their development work in such a complex environment. In this thesis, we build and implement a testbed for IoT and WSN developers to run their experiments in a controlled and realistic environment. The testbed comprises of heterogeneous embedded hardware which gives it a real world touch of an IoT ecosystem. In this testbed, users can schedule their experiments using the testbed management software, run them and log results at the end of the experiments. We have also developed an inexpensive tool for debugging IoT embedded hardware and the programs running on it. The results of our work are discussed and evaluated in the evaluation chapter of this report.

Keywords: IoT, Wireless Sensor Networks, heterogeneous, actuators, ecosystem, testbed.

Acknowledgements

We would like to take this opportunity to thank our supervisor, Olaf Landsiedel, for his friendly assistance, advice, and feedback throughout the process of writing this thesis. In the same vein, we thank Beshr Al-Nahas for opening his door to us every time we could go to him with our technical questions. We also would like to acknowledge Göran Stigler and Chalmers Robotförening (CRF) for helping us with the 3D printing and laser-cutting services respectively. THANK YOU VERY MUCH!!

Fahad Lafta, Coleb Mujurizi
Gothenburg, Sweden
December 2016

Contents

1

Introduction

A smart object is an electronic device that is capable of interacting with the environment and communicating with other devices. It has a power supply, microprocessor to perform computations, a sensor or an actuator for interacting with the environment, and communication device to communicate with other objects. When smart objects connect and exchange data over a network, they form what is termed as the Internet of Things (IoT). When such networks consist of sensors that are not necessarily smart objects, then they form a Wireless Sensor Network (WSN). IoT is a huge revolution and Cisco predicts that 50 billion devices will be connected to the Internet by the year 2020 [1]. Ericsson, a Swedish telecommunications company, predicts the number of IoT devices to quadruple in Western Europe alone between 2015 and 2021 [2]. The world will eventually turn into an ubiquitous computing arena with wireless sensors and Radio Frequency Identification (RFID) chips embedded in every object.

The impact of the IoT revolution is already being felt. Industries and homes are already registering major changes in efficiency and productivity. The transport sector is getting a new face of autonomous cars, connected street lights switch off to conserve energy when nobody is in the streets, automated traffic monitoring is controlling congestion, smart metering systems in cities and homes, waste management, entertainment, it is an endless list of possibilities [3].

It is therefore of a particular interest that the heterogeneous smart objects, software applications, and protocols which make up the IoT architecture are tested in a controlled and realistic environment before they are rolled out for all users. This controlled environment is known as a testbed. The work in this thesis has been to build and deploy an IoT and WSN testbed in the Computer Science and Engineering department building at Chalmers University of Technology. The testbed will facilitate the testing, debugging and evaluation of IoT and WSN applications and services before deployment.

1.1 Project specific goals

The main goal of this thesis is to build a testbed that consists of simple, off-the-shelf hardware as well as open-source software. This in turn consists of three main sub-goals:

(a) Develop a testbed management software that aids users in creating and managing experiments on the testbed. This is a an essential part of the testbed in order to make it available for users that do not have access to the physical network.

(b) Develop an affordable and easily available tool for debugging embedded hardware and software. In this testbed, embedded hardware and software refers to the IoT and sensor nodes, and the software that runs and controls these nodes.

(c) Deploy the physical network for the testbed. On this network is where all the experiments will take place.

1.2 Motivation

Building a testbed which consists of off-the-shelf hardware and open-source software serves the purpose of offering researchers and developers the possibility to build and deploy their IoT applications in a readily available testing environment. In addition to the possibility of building an own testbed, it also saves them the time that would be lost in preparing and setting up the testing environment. With the testbed already in place, researchers and developers can focus on other areas.

The testbed management software allows users to access the testbed and its services. This is very important for users that aim to run experiments without necessarily having to deploy their own testbed. The management software offers such users the possibility to run and manage their experiments on the testbed.

Developers find it expensive to purchase tools to use in debugging their hardware and software during development. Using easily accessible components, we develop a debugging tool which is much cheaper than the commercially available tools but serves similar purposes. Another advantage is that the same components that we use to make the debugging tool can be used for other purposes at the same time, for example programming nodes and obtaining logs from them.

There is need to set up all the hardware which the testbed management software controls in a systematic and networked structure. This is achieved through deploying a physical network that coordinates both the testbed management software and the hardware to function as a single platform for users to carry out their tests and experiments.

1.3 Problem statement

When working with off-the-shelf hardware, one is limited to utilising the built-in specifications of the hardware. In contrast, if the hardware is customised, then no such limitations exist. In our testbed, we entirely use off-the-shelf hardware and this limits us to what we can do with the hardware since we cannot modify their specifications.

The testbed management software functions as the central component of the testbed where all operations on the testbeds physical network, and the interaction with the testbed users are performed. Integrating the communication between the management software and the physical network is challenging because the different components of the physical network need to be coordinated in order to accommodate with the experiments the testbed users aim to run.

1.4 Report outline

Chapter 1 gives an introduction to the work that we have done in this thesis, defines the thesis goals, motivates them and clearly outlines the problem statement. Chapter 2 gives the necessary details about all the background technologies, both hardware and software, which we based our work on. In Chapter 3, we reference the most related work which has been done as far as development of IoT and WSN testbeds is concerned. We discuss in detail three projects which are our major inspiration for carrying out this work. We then compare our work with the state of the art. Chapter 4 focuses on the design of the major components in our testbed whereas Chapter 5 shows the implementation of the testbed's network, user interface and the testbed's debugging functionality. In chapter 6, we evaluate our results and discuss them. Finally, we present our conclusions and suggest areas for future work in Chapter 7.

2

Background

This chapter presents an overview of the core elements from where we build the foundation of this thesis work. These elements are in the form of different technologies from different vendors and are both hardware and software. We combine these elements in our work to come up with the answer to the problem in question. In this chapter, we point out a few general details about each of them to provide the reader with accurate yet simple understanding of the underlying technical work carried out in this research. Further details about how we use and integrate the selected hardware and software components in our work are found in Chapter 4 and Chapter 5 of this report.

2.1 Off-the-shelf hardware

In this section, we describe most of the off-the-shelf-hardware which is relevant to our work. Off-the-shelf is the term used to refer to products which are manufactured according to a certain standardised format and are readily available for use. Such products do not require any more custom development. We chose to use off-the-shelf products whenever possible because of the benefits that come with them, namely, reduced costs, high product quality, well documented for maintenance purposes among others.

2.1.1 Single board computers

Single board computers are small size computers which have the microprocessor, memory, input/output, and other ordinary computer features built on a single circuit board. These particular computers are available at a low cost, and are widely used in very many IoT and WSN projects today. There are many single board computers in the market today from different manufacturers. Examples of single board computers include the Raspberry Pi, Arduino, BeagleBone, Banana Pi, Intel Edison, Parallella, to mention but a few. Figure 2.1 is a Beaglebone Black, Figure 2.2 is an Arduino Uno, and Figure 2.3 is a Raspberry Pi Model B+ computer.

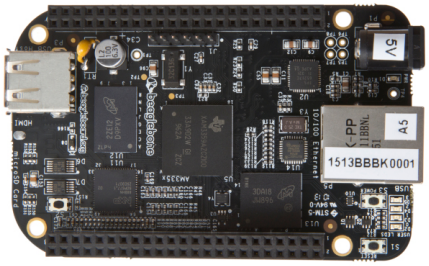


Figure 2.1: Beaglebone Black

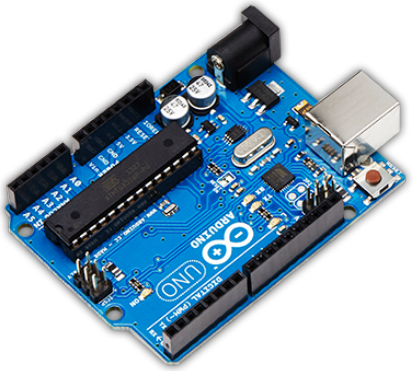


Figure 2.2: Arduino Uno

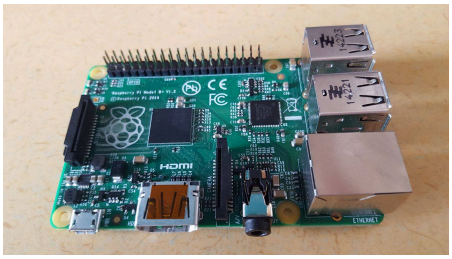


Figure 2.3: Raspberry Pi Model B+

Figure 2.4: MK20 USB board and BLE nano

2.1.2 Low power sensor nodes

These are micro-controllers which require low power in order to gather sensory information, process it, and communicate with other connected nodes in the network. Low power sensor nodes can run on battery for a very long time and can survive harsh environment conditions. Below are some examples of low power sensor nodes.

2.1.2.1 BLE nano

Bluetooth Low Energy Nano is a very low power and low cost development board manufactured by Red Bear Company Limited [4]. The Bluetooth Low Energy is a Wireless Personal Area Network (WPAN) technology that is growing rapidly in the IoT industry [5]. A BLE nano is as small as 8.5mm by 21.0mm and it supports a voltage range of 1.8V to 3.3V. It has a 12MHz Nordic nRF51822 System On Chip (SoC) with ultra low power consumption capabilities and a flash memory of 256kB. It is programmed through an MK20 USB board which connects to the computer through a USB port. The board also supplies power from the computer to the BLE nano through USB. Figure 2.4 shows a BLE nano with an MK20 USB board.

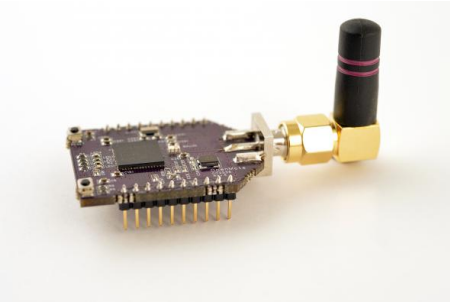


Figure 2.5: OpenMote CC2538

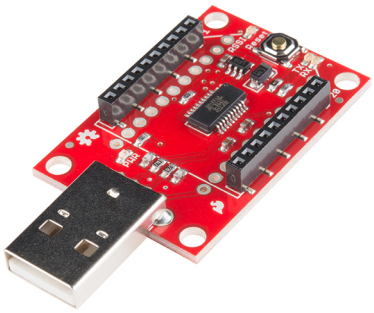


Figure 2.6: XBee Explorer Dongle

2.1.3 OpenMote-CC2538

OpenMote-CC2538 [6] is designed and developed by OpenMote Technologies, manufacturers of open-source hardware for the IoT industry. It has the following hardware specifications:

- A 32 MHz CC2538 SoC from Texas Instruments with 32kB of RAM, 512kB of programmable Flash and a JTAG debugging system.
- Four LEDS which are used for debugging
- Reset and user buttons for resetting and waking up the board from sleep mode respectively.
- An antenna connector for connecting an external antenna
- XBee header to connect the mote to an interface board. In this thesis, we use an XBee Explorer Dongle [7] from SparkFun Electronics as the interface board for the OpenMote-cc2538. Through USB, it provides the mote with power and a connection to the computer for programming and debugging.

Important to note is that the mote has a 10-pin JTAG header for debugging. JTAG and debugging are introduced later in Section 4.6.

2.1.4 Telosb

We use the MTM-CM5000MSP TelosB mote [8] which belongs to a family of open-source ultra low-power wireless sensor-motes from the University of California, Berkeley [9]. According to UC Berkerley, the TelosB is designed with three fundamental goals in mind, namely, minimum power consumption, ease of use, powerful software and hardware. This makes a TelosB a suitable platform for developers to deploy and experiment with in the field of WSN and IoT research. It consists of built-in light, temperature and humidity sensors. The mote runs on an MSP430F1611 processor from Texas Instruments with

Figure 2.7: The TelosB mote

48kB of Flash and 10kB of RAM. Texas instruments also supplies the mote with an IEEE 802.15.4 compliant CC2420 Radio Frequency (RF) chip [10] [11]. For connections, the TelosB mote features a USB interface, universal asynchronous receiver/transmitter (UART) and Serial Peripheral Interface (SPI). The mote is powered and reprogrammed with firmware through the USB interface. It has a 2xAA battery pack attached to it and batteries can be used as a source of power if it is not powered by a host computer through USB. Other features of the TelosB include three LEDs and two User and Reset buttons. The mote is fully compatible to Contiki and TinyOS, two popular low power WSN operating systems.

2.2 Open-source software & debugging

Customizability, free cost, and security are among the top benefits of using Open Source Software (OSS). We rely on open-source software tools in this thesis so as to maximise the benefits of its usage. In this section, we describe all the open-source software that we are using. We also introduce Joint Test Action Group (JTAG) debugging in this section.

2.2.1 Dropwizard

Dropwizard is a Java framework that combines almost all Java libraries to create a simple and light-weight framework for building RESTful web services. REST stands for REpresentational State Transfer and it is a standard software architecture of the world wide web which does not focus on implementation details but rather on a specific set of interactions between data elements and component roles. RESTful web-services are therefore such services which conform to the REST architectural standards. RESTful web services majorly depend on Hypertext Transfer Protocol (HTTP) as a communication protocol and deploy the usual GET, POST, PUT, and DELETE methods to retrieve web pages and send data to remote servers. In a RESTful setting, external systems are accessed as web resources and are identified by a string of characters known as Uniform Resource Identifiers (URI).

In this thesis, we choose to use Dropwizard because it offers a complete package with

out-of-the-box support for sophisticated configuration, embedded HTTP server, RESTful endpoints, built-in operational metrics and straightforward deployments. Dropwizard's bundle of light framework and libraries make it easier for a developer to concentrate on developing RESTful services in Java rather than wasting time on bringing them all together. This reduces maintenance burdens. In other words, ample time is invested in the essential complexity of the problem at hand rather than plumbing. The following are some of the examples of Dropwizard libraries:

- Jetty for HTTP service
- Jersey for building RESTful web applications
- Jackson for serialization and deserialization of JSON for REST services

2.2.2 Contiki

Contiki is an open source, highly portable operating system for the Internet of Things [12]. It is designed to run on a variety of tiny hardware devices (microcontrollers) with very small memory, processing speed, low power and bandwidth. It supports standards such as IPv6, IPv4, HTTP and other sets of lightweight low power wireless network protocols such as IPv6 over Low power WPAN (6LoWPAN), and the Constrained Application Protocol (CoAP). Contiki was first developed by Adam Dunkels in 2002 but other expert developers from all over the world have since then contributed to its further development. It is written in C language and provides a command-line shell with a set of commands which are used in programming and debugging Contiki hardware [13]. Applications can be compiled for any of the available Contiki platforms by using Contiki's build system.

2.2.3 JTAG debugging

Joint Test Action Group (JTAG) is a standard that was developed in 1985 to provide a technology for testing complex Printed Circuit Boards (PCBs) without the need to physically access the boards [14]. JTAG creates a direct connection into the CPU's internal debug logic hence facilitating the control of program execution on the embedded microcontroller. After tapping into the core of the embedded system's CPU, both hardware and software debug operations are possible. For example, JTAG can be used to set both hardware and software breakpoints, reset and initialise the target board, and upload firmware onto the board's flash memory. In an IoT and WSN development, developers need to interface with embedded processors of different architectures and this is where JTAG is needed. Using a special JTAG probe hardware or dongle, a connection from the target system/device to the host computer is established either through USB, parallel port or Ethernet. The setup is as shown in Figure 2.8. In our thesis, we implement JTAG debugging on the OpenMote [6] board without having to buy a special JTAG probe hardware. We customize the Raspberry Pi's GPIO pins and the OpenOCD [15] library to achieve the same JTAG debugging functionalities using GDB.

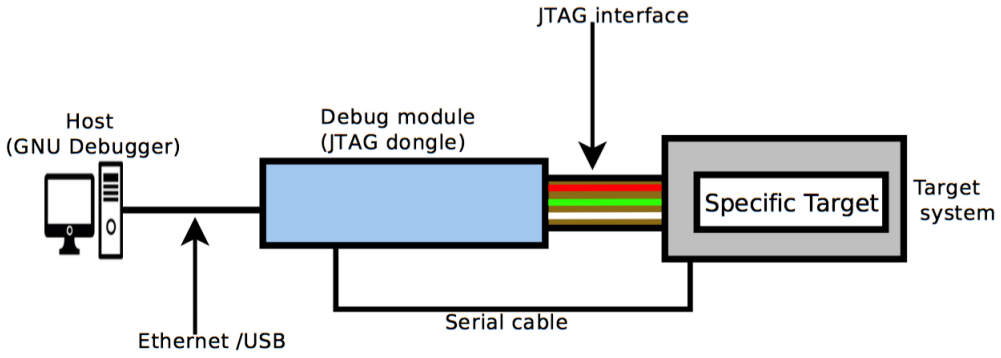


Figure 2.8: Setup of JTAG debugging

2.2.4 OpenOCD

OpenOCD stands for Open On-Chip Debugger and is an open-source software that provides in-system programming, debugging and boundary-scan testing for embedded JTAG IEEE Std 1149.1 compliant [14] target devices. It was first created by Dominic Rath as part of his diploma thesis in 2005 at the University of Applied Sciences, FH-Augsburg, Germany and now the source code is available through the GNU General Public License [15].

OpenOCD assists in performing debugging, in-system programming and boundary scan testing on the target devices with the help of the debug probes or debug hardware as mentioned in Section 2.2.3. The hardware can either be a JTAG debug hardware or Serial Wire Debug (SWD) hardware. The former supports both debugging and boundary scan operations whereas the latter only supports debugging. It acts as a middleware running between the debugger and the JTAG adapter. It translates GDB commands and sends them to the JTAG adapter [16].

In our testbed, we use OpenOCD in preparing the Raspberry Pi to use it as a JTAG debugging adapter. We use the library to customize the Pi's GPIO pins for JTAG debugging. It also acts as the GDB server and lets the gdb client application communicate with it through a remote TCP port.

2.2.5 GDB

GNU Project Debugger (GDB) is an official debugger for the GNU operating system [17]. It is an open-source software that is used to trace and alter a computer program's execution. It enables one to see what is taking place "inside" a program while it executes [18]. With GDB, programmers have full control of the program's execution for instance, they can specify the locations where they want their programs to pause before continuing to execute by setting breakpoints, they can set watchpoints to monitor programs when-

ever expression values change, halt and resume program execution, etc. GDB debugs programs written in C, C++, Java, Ada, Pascal and several other languages. GDB has two modes, the server and client mode. The latter runs on the host computer and the former runs on the embedded target system. GDB client acts as a user interface whereas GDB server acts as an application controller and they both communicate with each other through a remote TCP-based protocol or serial connection [16]. In our work, we use a GNU Debugger for the ARM EABI (bare-metal) targets [19] such as the one described in Section 2.1.3 to perform debugging. We describe all the detailed procedures in Section 4.6 and Section 5.3 of this thesis.

3

State of the art

In this chapter, we go through five of the already existing testbeds. Indriya [20], FlockLab [21], and TWIST [22], are the three testbeds which are major inspirations to our thesis. We therefore discuss them in more detail than the other testbeds, namely FIT IoT Lab [23] and SmartCampus [24]. After describing the testbeds, we proceed to compare them with each other in regards to their key features. Finally we compare our own testbed's design goals with theirs.

3.1 Major inspiration

In this section we describe three testbeds which are a major inspiration to our testbed. These testbeds have features and properties that are available in our testbed, and that is why we describe them in a more detailed way than the other testbeds in the next section.

3.1.1 Indriya

Indriya is a testbed that is deployed at the School of Computing of the National University of Singapore (NUS) [20]. The testbed is developed by Manjunath Doddavenkatappa, Mun Choon Chun and Ananda A.L at the afore-mentioned school. Indriya has been available for use within the university since April 2009 and publicly available since December 2009. Since then, researchers and students from more than 35 universities around the world have used the testbed to conduct IoT and WSN experiments.

According to the testbed's creators, Indriya is "a large-scale, low-cost wireless sensor testbed". It is designed in such way that its deployment and maintenance costs are reduced. Its USB infrastructure, which powers and programs the sensor nodes, reduces the deployment costs. Without such infrastructure, the nodes would need adapters to draw power and that is expensive. More so, when the USB infrastructure is used to

power the nodes, it reduces maintenance costs. Power adapters are usually not designed to have long-term use. The testbed administrator has to replace these adapters whenever they fail. The larger the testbed is, the more costly and time consuming it becomes to replace these adapters.

Indriya uses TelosB motes. TelosB motes are a desirable choice for a testbed like Indriya because their programming and powering happens through USB. More than half of the sensor nodes in Indriya deploy different types of sensors, for example Passive Infrared (PIR), magnetometers and accelerometers.

The deployment of the Indriya testbed spans over three floors hence making the wireless connectivity among the nodes three-dimensional. This opens doors to possibilities for the testbed users to experiment with protocols that rely on connectivity as well as placement, for example, geographic routing protocols. In addition to using passive USB cables to span the building, Indriya also deploys active-USB cables which can sustain a signal up to 25 meters. Another factor that contributes to the geographical span of Indriya testbed is that it groups the nodes together into clusters, where each cluster-head can take up to 127 nodes. A cluster-head in this case is a Mac Mini PC.

Indriya users create, schedule and monitor jobs using a user interface that offers web-access to the nodes of the testbed. To execute a job, a user has to first create it and upload the necessary software (compiled code for TelosB), and then schedule the job and specify its duration. The duration has to be within the time that the testbed is allocated to the user. Indriya does not guarantee that the nodes will be programmed simultaneously, which can be desirable for experimenting with certain types of WSN and IoT applications. Indriya offers its users the possibility of monitoring their executing jobs in two ways. The first one is by presenting the data the user's software dumps on the USB port; the testbed presents the data after the job finishes. The second one is by monitoring the USB-dump in real-time, by accessing nodes' serial forwarder through a TCP connection.

3.1.2 FlockLab

FlockLab is a testbed deployed at the Swiss Federal Institute of Technology in Zurich, also known as ETH Zurich [21]. It offers the possibility to observe its nodes closely. FlockLab provides its users with the service to study several types of behaviours which are not necessarily connected to the serial port. These services include GPIO tracing, GPIO actuation, power profiling, adjustable voltage supply and serial I/O.

The FlockLab architecture consists of observers, targets, and servers. In the centre of the architecture lies the observers, which are platforms that are connected to targets at one end and connected to the server on the other end. They act as the middle-men between the server and the targets. Targets are platforms or devices that run the test software which the testbed user uploads. The server coordinates and synchronises the

operations of all the observers that connect to it. The server also handles the interaction with testbed users as well as storing results of the tests which have finished executing on the testbed.

FlockLab supports heterogeneous target nodes, which means that users can run their tests on different platforms. The target nodes that FlockLab deploys are Tmote Sky, Tinynode 184 [25], Opal [26], and IRIS [27]. The observer hardware is more special for FlockLab and is based on an assembly done on a custom-designed Printed Circuit Board (PCB).

FlockLab comes with a standard testbed service, namely serial I/O. Through serial I/O, a user can read as well as inject data through the serial port of the target. Besides serial I/O, FlockLab offers a set of services that help a user to observe and monitor closely the behaviour of the targets while running the users tests. This is a very desirable feature for certain users that aim to not only obtain the results from the targets but also to study the behaviour of these targets while the tests are under execution. FlockLab offers GPIO tracing, where a user can monitor and trace the level exchange of 5 target pins in a frequency of up to 10 kHz. FlockLab also offers GPIO actuation where a user can clear, set and, toggle up 3 target pins. Using GPIO actuation, users can set triggers at certain times or periodically in order to create more controlled experiments that trigger event(s) at certain period(s) of time.

FlockLab also offers services that monitor the power behaviour of target nodes. An example of such services is power profiling where a user can sample the current amount of electricity the targets are using. Another such service is adjustable voltage supply where a user can adjust how much voltage the target supply gets. Both these services, and also the services that we mentioned in the previous paragraph offer great help for users aiming to study the behaviour of the targets closely.

In order for FlockLab to function properly as a testbed, it has a set of the following back-end servers:

(a) a Network Time Protocol (NTP) server for time synchronisation of all observers and for providing a precise time reference across the testbed.

(b) a web server that handles the interaction with users and that is where they can schedule and configure their tests.

(c) a test management server that handles the starting, running and finishing of the tests

(d) a database server for storing test information as well as user-specific data such as login information.

(e) a monitoring server that monitors other servers, networking components as well as observers. In case this server discovers any abnormal behaviour, it notifies the admins of the testbed.

FlockLab users can use its user interface in order to create, configure and schedule jobs. A user uploads, besides the needed software, an XML configuration file where he/she specifies the configuration of the test he/she is attempting to create. After the user creates the job, the testbed assigns it a time slot in the schedule and executes it when that time arrives. When the job finishes executing, the testbed sends the results to the user by email. The user can also download the results using the web interface.

3.1.3 TWIST

Standing for TKN Wireless Indoor Sensor network Testbed, TWIST is a testbed that is deployed at the technical university of Berlin. The name is derived from the fact that the testbed has evolved from an earlier testbed deployed at TKN (Telecommunication Networks group) at the technical university of Berlin. Authors of TWIST describe it as a "scalable and flexible testbed architecture" [22] . It comes with features and services that make it not only flexible and scalable, but also dynamic when it comes to hierarchy and topology. These features include active power supply, support for creation of flat and hierarchical sensor-nodes among others.

TWIST, like Indriya, relies on a USB infrastructure for powering the nodes. This has made the active power supply control of the nodes possible hence allowing a user to shift between different powering alternatives for the nodes. For example, if the nodes can draw power from a battery as well as through USB, then the user can shift from USB-powered mode to battery-powered mode by cutting off the power supply from the USB-port and vice versa. Powering by USB or battery forms a subject for experimentation for some users. This is a desirable feature available in TWIST. With this feature, the testbed can power off nodes that are in an idle-state for example. The users of the testbed can use this feature to inject node failure in a controlled way and then observe the behaviour of the system and how it accommodates with that failure. A user can also utilise powering off the nodes to control the topology in the network, where he/she can power some nodes off in order to achieve a desired topology more suited for a certain type of experiment. This is why the support for active power supply control of the nodes is a feature that makes TWIST a flexible testbed when it comes to topology.

TWIST does not only support dynamic topologies but also dynamic hierarchies. The users of the testbed can choose between having a flat sensor network or a hierarchical one. This feature is available through the utilisation of super nodes, where several nodes connect to a super node and then the super node in turn connects to the server. If the user is looking for a flat architecture, meaning that application or test will run only on the low-end sensor-nodes, in that case the super nodes act as local servers for the sensor-nodes. If the user is looking for a hierarchical sensor network, meaning he/she aims to execute part of the application on low-end sensor-nodes and another part on high-end senor nodes, then the super nodes act as high-end sensor-nodes and the user achieves the desired hierarchical network.

TWIST supports heterogeneous nodes. It supports any node as long as it has certain capabilities that makes it compatible with the TWIST architecture. These capabilities include mainly the ability of the node to draw power, receive programming instructions as well as communicate through USB. This requirement is mainly due to TWIST's USB architecture, where the testbed performs all operations on the nodes through USB.

TWIST is a scalable testbed, which is a property it achieves due to several factors. First of all, it makes use of super nodes which helps not only in achieving dynamic hierarchies but also in making the testbed more scalable. If the testbed does not use super nodes, then there will be a limit on how many devices that can connect to the main server directly. Using the super nodes makes that limit increase considerably, as each of these nodes acts like a server and then all of them connect to the main server. Another factor that makes TWIST scalable is that it utilises off-the-shelf hardware as well as open-source software, which furthermore makes it cost-friendly and replicable.

In order for TWIST to function properly, it has a back-end in the form of a server as well as a control station. This back-end is responsible for the interaction with the testbed users on one end, and with the super nodes on the other end. The server and the control station form together a backbone that all the super nodes connect to. The control station decreases the workload on the server by starting a separate thread for each super node that connects to the backbone.

TWIST users interact with the testbed using a user interface. The users utilise a GUI toolkit to schedule and run experiments on the testbed. When an experiment finishes, users can use the toolkit to release the nodes so that they become available to other experiments.

3.2 Other testbeds

In this section we furthermore describe 2 testbeds that exist today even if they have not been a direct inspiration for our testbed.

3.2.1 FIT IoT-LAB

The FIT IoT-Lab is a testbed deployed at six sites across France and it is a testbed available for experimenting with large-scale wireless IoT technologies [23]. What makes IoT-LAB suitable for large-scale IoT experimenting is the fact that it consists of 2728 low-power wireless nodes as well as 117 mobile robots. In addition to that, the testbed is deployed at 6 sites across France where these sites are interconnected. This makes IoT LAB not only a large-scale testbed when it comes to the number of nodes in it, but also considering the geographical area it covers.

IoT-LAB supports heterogeneous nodes, which is always a desirable feature as this allows experimenting with a wider spectrum of IoT and WSN applications. The IoT-LAB

testbed widens that spectrum even more by offering a number of mobile nodes in addition to the static ones. The testbed users can utilise these mobile nodes in experimenting with IoT applications where mobility is a focus. For example, in wireless networking protocols, mobility is usually an issue that affects their performances. The mobile nodes are in the form of robots and users have the ability to move these robots during experiments.

The nodes in IoT-LAB consist of three components, namely the open node, the gateway, and the control node. The open node is the low-end device that the user programs. The gateway node is a small computer that connects to the open node, the control node, as well as to the backbone. It monitors and forwards the serial activity of the open node to the back-end servers. The control node is the one that coordinates programming on the open node.

IoT-LAB is an open testbed, meaning that it offers the possibility to the user to have direct access to the nodes so they can develop their own firmware on them. Therefore, the testbed provides a set of software for the user to use in the development process. This software includes low-level drivers and libraries, which offer the user access to nodes' hardware. Another part of the user software is the embedded operating systems; IoT LAB supports several IoT operating systems such as RIOT [28], FreeRTOS [29], Contiki [12] and TinyOS [30] among others.

The users of IoT-LAB interact with the testbed through either its web interface or its REST API. In order to launch an experiment, the users start by reserving node(s) for a certain amount of time. The user then proceeds to make the configuration of the experiments regarding how the nodes will get power, specifies what monitoring tools to utilise, and uploads the necessary firmware. The scheduler then starts the experiment as soon as the resources become available, or at a certain time depending on what the user has specified.

3.2.2 SmartCampus

SmartCampus is a testbed environment deployed at the university of Surrey in the United Kingdom. It is a user-centric testbed which offers interaction with real end users during experiments [24].

The SmartCampus testbed offers a realistic IoT experimentation environment through not only allowing end-users to interact with the experiments, but also through its deployment in a real office building. The latter offers an essential part of the Smart Cities, where IoT applications will be mostly applicable. Giving the opportunity to experiment with such environments gives the IoT experimentation a much desired realistic dimension.

The end users contribute mainly to the realism of the environment by consuming the data that the experiment generates. They also contribute by being the source of generated data, for example the data which is generated by motion sensors. A testbed user can

then use this data to measure the IoT applications performance for example. Other researchers can also make use of this type of data to analyse human behaviour in general.

The architecture of SmartCampus consists of three tiers. The first one is the server tier which handles the back-end functionality as well as interact with the testbed users. The second one is the embedded Gateway tier where it forms an infrastructure that connects the IoT nodes to the back-end. The third is the IoT tier where the IoT nodes are. SmartCampus supports heterogeneous nodes, which is also an important factor in making IoT experimentation more realistic and flexible.

SmartCampus utilises smartphones as well as smart displays to enhance the user-centric experience in the testbed. The smartphones help in both the sensing of the users and helping with their interaction with the testbed. The smart displays enhance further the interaction between the users and the testbed inside the building where the testbed is deployed. We have not seen these types of hardware components in the previous testbeds, which shows how SmartCampus deviates from the other testbeds when it comes to the level of involvement and interaction of the end-users.

The testbed users or experimenters can access the services of SmartCampus through a graphical user interface called TMON. The interface allows the testbed users to create and configure experiments, as well as assisting them during the experiment execution and data analysis when the execution is completed.

3.3 Testbed comparisons

We have described five of the testbeds that are deployed and utilised in the IoT community today. Now we proceed to compare these testbeds in order to get the bigger picture (what features these testbeds have in common and how they differ). This comparison will help us to realise where our testbed fits with these testbeds.

All the testbeds which we have mentioned have support for heterogeneous nodes. This is a very desirable feature when building a testbed as it widens the spectrum of the experimentation of IoT applications. Node heterogeneity is a way to achieve a more realistic testing environment for IoT applications, as these types of applications will most likely apply in an environment where it makes use of multiple types of IoT nodes.

In terms of re-usability, Indriya and TWIST differ from other testbeds. The two testbeds utilise off-the-shelf hardware as well as open-source software which makes them more reusable than the other testbeds. The re-usability feature is important in order to make a testbed easily replicable, which is a desirable feature in a field of research as relatively fresh as IoT. If a testbed is easily replicable, then other researchers can recreate it and do further research on it. This feature is not very important for the testbed itself, but it is definitely a feature that makes the testbed more usable in the context of research than testbeds that are not easily replicable. The other testbeds do use off-the-shelf hardware

and open-source software also, but not to the same extent as Indriya and TWIST which makes them not as easily replicable.

Indriya as well as TWIST differ from the other testbeds in the sense that they make use of a USB infrastructure in order to power the nodes. The utilisation of such infrastructure in powering the nodes has the advantage of eliminating the maintenance costs and time spent on replacing faulty power adaptors, which are usually not suitable for long-term usage. Building such an architecture on the other hand is quite costly and time consuming, especially if it will integrate with the infrastructure of the building. This works great for Indriya for example because the testbed utilises a USB infrastructure that has already been there before.

In one way or the other, all the testbeds that we have mentioned have an architecture that consists of 3 tiers. The first tier is the back-end, which is usually in the form of a server that hosts a database to store the information about the testbed experiments as well as the users of the testbed. The server also acts as the entry point of the testbed and handles the interaction with users. The second tier consists of hardware that connects the first and the third tier together. Components in this tier coordinate the programming of the nodes in the third tier with the user instructions received at the server. The second tier also forwards the data that the IoT nodes dump to the server where the latter stores it, so the user can later on retrieve the data from the server. The third tier consists of the IoT nodes where users run their programs and experiments on. How these tiers work and what components they consist of differs from one testbed to the other, but the architectural principle is the same.

FIT IoT-LAB addresses the mobility issue which the other testbeds do not address. It is the only testbed that uses both static and mobile nodes. The mobile nodes are in the form of robots where the experimenter has control over them during the experiment. This makes the testbed more suitable for testing certain types of IoT applications and protocols where mobility affects the performance. Mobility is an important aspect of IoT experimentation, but implementing it is costly. Otherwise, more testbeds would implement that feature as it is very relevant and desired in the field of IoT.

All the testbeds have a user interface in the form of a web interface where the testbed users can create, and manage their experiments. The interface acts as an access point that connects the users with the testbed. The procedures that the users need to follow to create and manage their experiments deviate in the different testbeds. One testbed may ask the user to specify the configurations of the experiment in an XML configuration file (FlockLab), while another testbed asks the user to set the configurations manually (TWIST and Indriya). The user needs to start the experiment when the resources are ready (TWIST) while he/she can start the experiment automatically once the resources become ready (FlockLab, Indriya), the latter is known as autonomous operation. The instructions may differ, but the role of the user interface is the same for all testbeds.

In addition to the user interface, FIT IoT-LAB offers a REST API that users can use to create and manage their experiments. This API contributes greatly for users aiming to automate their interaction with the testbed server. Through this automation, users can access the testbed services using the API without having to use the web interface manually, but rather by writing and executing a script, which is something that advanced users feel more comfortable about.

3.4 Our contribution

After describing some of the testbeds that researchers utilise today and comparing them with each other, it is now time to describe our testbed that is the focus of this thesis work. We will talk about what has inspired our testbed and we will then proceed to describe its properties and features.

The first step that we have taken in this thesis is to read about existing testbeds in order to understand this area of research on one hand, but also to look at actual implementations in order to understand what is missing and from there come up with what we can contribute with. It did not take us long before we realised that each testbed that we studied had certain feature(s) that we preferred and appreciated more than others. This raised the question: would it not be great if we could put all the features that we prefer the most in one testbed?

The main ingredients of our testbed is simplicity and re-usability. Our testbed consists of simple components, both hardware and software. This helps in saving time for both users of the testbed and personnel maintaining it. In addition to the components being simple, the hardware is off-the-shelf and the software is open source. This makes our testbed reusable and replicable; we see this testbed as a research project and we welcome other researchers to replicate and develop it further for the benefit of the IoT and WSN industry.

Our testbed supports heterogeneous IoT-nodes. We do not want the testbed to support only one type of nodes, because it severely limits the level of IoT experimentation for our users. In addition to heterogeneous IoT-nodes, some nodes are even equipped with JTAG debugging capabilities which allow the testbed users to debug their software running on the nodes.

The testbed supports automation as well, the REST API makes automating the interaction between the users and the server in order to create and manage experiments on the testbed possible and straightforward.

Last but not least, our testbed supports autonomous operation. Once a user schedules an experiment, it will start without the user having to start it manually. The user has the option to either schedule an experiment as soon as possible (as soon as the resources it needs become available), or schedule an an experiment to begin at a certain time.

4

Design

In this chapter, we show a detailed design of our testbed and all its components. From the design of the whole system, we break it down from the server to the client node describing how each component is supposed to look and function.

4.1 System overview

This is the general physical representation of our testbed, as Figure 4.1 illustrates. **SBC** represents an embedded small-size and affordable computer (single board computer) with sensor motes **A**, **B**, and **C** connected to it. Both the sensor motes and the single board computer make up a client node. A central server controls the entire testbed. It connects to the network with all other client nodes. Through a user interface, the server creates a platform for the testbed user to communicate with the client nodes.

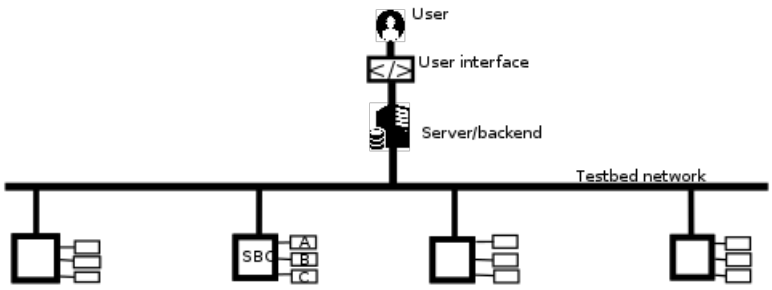


Figure 4.1: System overview of the testbed

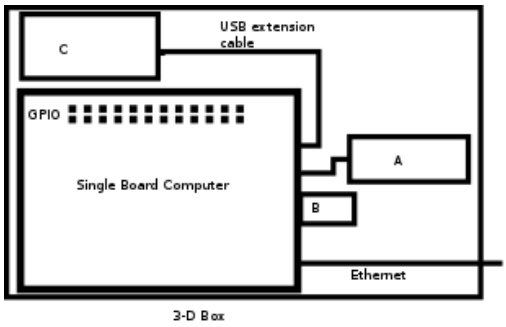


Figure 4.2: Client node design

4.2 Client nodes

Figure 4.2 illustrates the physical organisation and arrangement of a client node. As we have previously mentioned, a client node is made up of single board computer and sensor motes A, B, and C connected to it. They connect to the same network as the testbed's server through an Ethernet link. The architecture of the network is a client-server architecture where all the nodes in the testbed connect to one central server as shown in Figure 4.1. We choose to use different models of sensor motes from different vendors to provide support for heterogeneous hardware in our testbed. Connecting the motes to the single board computer through USB serves two purposes. First, to avoid the burden of powering them separately as this will make the system bulky with the increase in number of power chords. Second, to make it easy to program and control through the computer's USB ports.

The single board computer hosts the firmware before uploading it to the motes as well as providing a programming environment for them. In our testbed, we will install the client nodes in different locations of the building. JTAG debugging will be available on one of the sensor motes. The design of JTAG debugging is explained in Section 4.6 of this report. We will program and control the other motes through the single board computer's USB interface and not through the JTAG interface. Connecting the motes directly to the computer's USB ports and next to each other results into a very congested and undesirable design. We come out with a layout design of the motes that makes them sit comfortably in the box and each maintaining its space. Good spacing of the motes eliminates signal interference from each other and maintains a neat design. We use USB extension cables to connect the motes to the single board computer while maintaining a neat layout design as illustrated in Figure 4.2.

4.3 3D printed boxes

Client nodes need to be housed and protected from external damage and dust. For this reason, we will design boxes for them as illustrated in Figure 5.1. The single board computer and the sensor motes need to sit firmly inside the box. It is important that

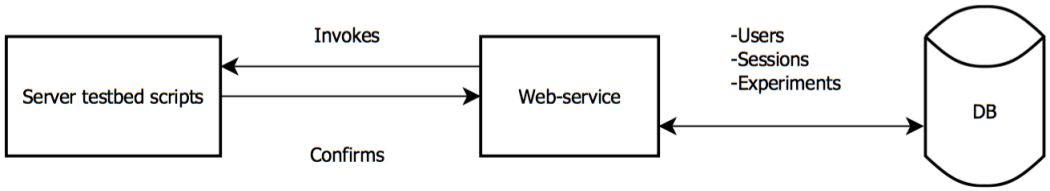


Figure 4.3: The back-end components, consisting of the web-service, a database and the server testbed scripts. The web-service is the central component that holds the whole back-end together

the design of the boxes provides easy access to the SD card, Ethernet connector and micro-USB power port. Because we need to physically monitor the status of the client nodes, the cover of the box has to be transparent and must be easy to remove and place back. This allows the testbed administrator to check if the nodes are alive or not by just looking at the nodes' LEDs through a transparent cover.

4.4 Testbed network

Whenever we mention the testbed network, we will be referring to a Local Area Network with access to the Internet. Both the central server and the client nodes connect to this network through Ethernet cables. Although security, signal interference, and connectivity speed are not our major concerns in this thesis, we still choose Ethernet over WiFi because the environment of our testbed favours an Ethernet setup. Client nodes acquire IP addresses from the server. We explain how the server handles the IP assignment of client nodes in Chapter 5 of this report.

4.5 Server and back-end

In this section, we take a closer look at the server of the testbed, and at what it hosts and what role it plays in the testbed. The server has two main functionalities; first is to act as an access point to the testbed users in order to interact with the testbed services through the user interface. Second is to host the back-end components which communicate with each other to handle the creation and management of the user experiments as shown in Figure 4.3.

DB: The database of the testbed, which stores information about users of testbed, as well as session data for users that are currently logged in to the testbed. In addition to that, it stores information about the created, scheduled and completed experiments of the testbed users.
Server testbed scripts: A set of scripts that handle the interaction between the server and testbed client nodes. These scripts coordinate for example the operations that are needed in order to prepare an experiment, run it and clean up after it completes.
Web-service: The central component for the interaction between the testbed users

Figure 4.4: Migration of experiment entities between different tables in the database

and the client nodes. The web-service receives instructions and commands from users, communicates with the database and calls the server testbed scripts with the proper parameters that match the users instructions.

Now we proceed to describe these components further, in order to get a deeper understanding of the role each one of them play in the back-end of the testbed.

4.5.1 Database

In every system that requires having a user account in order to access some services, a database becomes an essential component. By connecting the system to a database, the data will be stored externally outside the kernel of the system. This means that the system becomes more robust, as any failure in the system will not cause any loss in the data in the database. When the system recovers from the failure, it can restore this data from the database. Robustness is extremely important in systems that have a long-term usage, which in this case our testbed has. This is why a database is one of the main components that build the back-end of the testbed.

The type of information that the testbed stores in the database falls in three main categories. The first category includes user data; containing information about the users and their login credentials. The second category is login data: containing session information about users that are currently connected to the testbed. The last category, which is the largest, contains experiment data. This includes experiments that users create, experiments the testbed schedules as well as experiments that complete their execution.

When a user creates an experiment and specifies the starting time, it goes through certain stages, or a life-cycle. In the database, this translates into the experiment information moving between different tables within the database depending on what stage the experiment is in, see Figure 4.4.

Pending jobs: experiments or jobs that the users have created and which the testbed has scheduled. These jobs are awaiting execution
Current job: current running job on the testbed. This table has only one entity
Completed jobs: jobs that have completed their execution.

Entities of experiments in the database can migrate between the different tables. This migration depends on the current status of the experiment. The only component in the

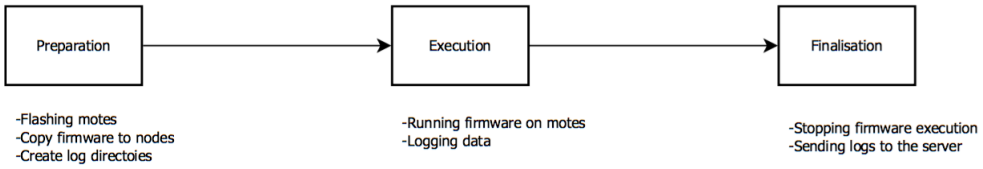


Figure 4.5: Client node phases. A client-node goes through these phases in connection with every experiment on the testbed

back-end that connects to the database is the web-service, since it handles, among other functionalities, the interaction with the testbed user. The web-service checks all the user instructions and commands against the database to guarantee that only testbed users can access the services of the testbed and ensure that they do not interfere with each others' experiments.

4.5.2 Server testbed scripts

The management of experiments on the client nodes is a systematic task that needs a great amount of accuracy and coordination to successfully run an experiment on these nodes. In order to accomplish this, a set of scripts on the server becomes an essential component of the back-end. The main role of the scripts lies in guiding the client nodes through the different phases that these nodes go thorough in order to successfully run an experiment on the testbed, see Figure 4.5.

Preparation: setting the client nodes for a new experiment. This includes for example flashing the motes prior to reprogramming them with the user firmware, copying user firmware and experiment configuration to the client nodes. In this phase lies also creating a directory for where the logs will be stored after the experiment completes.
Execution: running the firmware of the experiment on the motes, and logging the data that the motes dump on the serial port along the away. Prior to this phase, the client nodes have already received the firmware of the user. In this phase, they program the firmware on the motes the user asks for and logs the data the motes dump during the execution.
Finalisation: the final phase in the life-cycle of an experiment on the testbed. In this phase, every client node in the experiment stops the execution of the firmware, and the server receives the logs from the client nodes and stores them.

The testbed goes through the phases in Figure 4.5 with every experiment it runs, but with different parameters. These parameters make the user configuration of his or her experiment. It is therefore very important to invoke the server testbed scripts with the proper arguments that the user specifies. This is a task that the web-service handles.

In order for the scripts on the server side to do their job, they invoke and communicate with another set of scripts on the client nodes. The server scripts handle the coordination

of the operations that the client nodes perform on the motes. We will discuss further more details about the scripts in the next chapter.

4.5.3 Web-service

The web-service is the core of the back-end of the testbed. It is the software component that connects to the other back-end components namely; the database and the server testbed scripts. In other words, it serves as the "brain" of the back-end; it makes the decisions regarding what to schedule and when to do so. It connects the memory (the database) to the rest of the body (the client nodes). The web-service also handles the interaction with the testbed user. It receives instructions and commands from the users, takes the proper action that corresponds to them.

The web-service makes use of the REST architecture that we have mentioned earlier on in Section 2.2.1. Therefore, before we proceed to describe our web-service, we will take a moment to talk about REST architecture first.

4.5.3.1 REST architecture

The term REST stands for REpresentational State Transfer and was introduced in the year 2000 by Roy Fielding in his doctoral dissertation titled Architectural Styles and the Design of Network-based Software Architectures [31]. He developed the REST architectural style in conjunction with his authorship of the Hyper Text Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI). This architectural style defines a general and generic interface for the web, where different components can interact with each other and across different platforms. The architecture builds on the principle that a service on the web consist of components, where each component has a unique address, or URI. The client interacts with these components through requests, where a request consists of the component's address as well as the information that the component needs in order to process the request

The REST architecture comes with a set of restraints that are important to us and that motivate our choice of the architecture. These restraints are the utilisation of the client-server model as well as the stateless aspect of the communication. The client-server restraint separates the responsibilities, or concerns, between the client and the server. For example, by separating the user interface from storage and scheduling, we can make the user interface more portable across different platforms. The stateless constraint defines the nature of the communication, which should be stateless. In other words, when the client sends a request to the server, the request should contain all the information that the server needs in order to process the request. This simplifies the server components, as they do not need to keep state, and this makes the service more scalable due to the fact that the components are independent of each other.

The testbed web-service builds on a perspective that the REST architecture is derived from, namely Starting with Null-style. With this perspective, the designer of an archi-

tecture starts with nothing, or a blank state, and builds it up by adding components till it satisfies the requirements and needs for the systems it aims to serve. This perspective has been of great importance when building the web-service, as it allowed us to add components gradually while maintaining a running and functioning web-service for the testbed.

4.5.3.2 User interface

As we have mentioned earlier, the web-service serves as the brain of the testbed. The service comes with a user interface that the testbed user utilises in order interact with the testbed as illustrated in Figure 4.6. The interaction between the testbed user or the client, and the server or web-service, builds on the REST architecture as we have mentioned before. The web-service consists of several components, where each component represents a functionality or a feature that the testbed user can interact with. For example creating a new experiment is a component, stopping an experiment is another component. Retrieving logs obtained from a completed experiment is also a component as well as logging in and out from the users account, and so on. Each of these components has its own URI as well as one or a set of parameters that it expects in order for a resource, or a component, to do its job. The communication is therefore stateless, which serves greatly when it comes to the portability of the user interface.

The combination of the URIs of the different resources, their input parameters as well as the values they return build together an API or a REST API in this case, that serves as the user interface for the testbed. The API offers two main advantages. Firstly, it offers support for automation. Automation has been one of the goals of this thesis work where testbed users will be able to interact with the web-service of the testbed using scripts. The API offers this support as users can easily write scripts according to the specifications of the API in order to interact with the web-service in the way they intend to.

The second advantage of the API is that it is generic and can be utilised to build different types of user interfaces and on different platforms. For example, the API can be utilised to build a web interface where testbed users can interact with the web-service using a web browser. The same API can be utilised to build applications, such as mobile apps, where these applications also use the same API to interact with the web-service, and so on. The API serves as a generic interface, and this is one of the great principles that the REST architecture builds on. It offers a great level of flexibility as well as scalability, which is a very desirable characteristic on services that are available on the web.

4.6 JTAG Emulator Design

In Chapter 2, we introduced JTAG debugging and its role in the development and programming of embedded systems. The role of JTAG debugging is even bigger in embedded micro-controller devices such as wireless sensor motes. In our testbed, some sensor motes

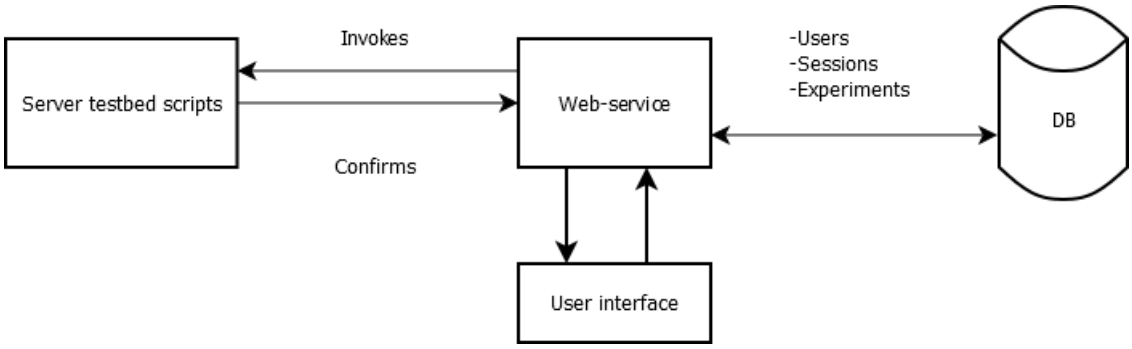


Figure 4.6: The user interface is utilised by the testbed user in order to interact with web-service.

have support for JTAG debugging while other motes do not. In an embedded system development environment, JTAG debugging is a suitable and efficient means of debugging concurrent programs with real-time constraints compared to the traditional "printf" debugging technique. The mote's JTAG interface is used to create a direct connection between a computer (single board embedded computer in this case) to the internal logic of the mote's CPU as illustrated in Figure 4.7. This design presents the following advantages to embedded system developers or testbed users:

- Having full control of program execution on the mote. For example, the ability to start a program, step through the program code, set breakpoints and checkpoints, view a specified line or block of code, halt and resume program execution, among other features.
- Being able to easily reset and initialise the mote and upload firmware onto its flash memory.
- Being able to test and debug the operations of the device without need for physical access to the mote. For example, it is easy to tell if some of the embedded device's electronic components are not functioning because they are broken. This is termed as boundary scan testing and it is one of the major importances of JTAG debugging.
- Some motes or micro-controllers have backdoors disabled by default and cannot be reprogrammed before enabling the backdoors. A JTAG debugging module is required to enable the backdoor. This is among the reasons why we need JTAG debugging.

On top of the above benefits, we eliminate the use of commercially available debug modules and replace them with an inexpensive custom-designed JTAG emulators. The cost of one JTAG debug module/ hardware is three times as expensive as our design. Furthermore, using the classical "printf" debugging technique does not offer all the above mentioned advantages and that is also the reason why we choose the JTAG emulator

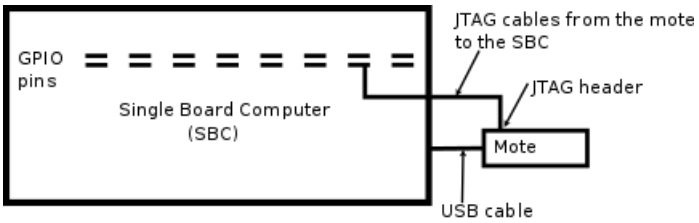


Figure 4.7: Design of the JTAG emulator showing connections from the mote's JTAG interface to the single board computer's GPIO pins.

design. However, the motes which do not have JTAG interfaces are going to be connected directly to the computer via USB ports as shown in Figure 4.7.

5

Implementation

This chapter describes how we setup the different components that make up our testbed. First, we describe what the testbed network consists of and how we set it up. We then explain the server configurations and testbed scripts which do most of the automation in the testbed. We also lay down the type of database management software that we use, and which tables and records it mainly stores. We then show the implementation of the RESTful web service which is one of the major features of our testbed. Finally, we explain the different libraries and tools that we use to implement JTAG debugging and how the implementation is done.

5.1 Testbed network

The testbed network consists of one central server running Ubuntu Linux distribution, and 25 Raspberry Pi computers running Debian Linux. From all the available choices in embedded single board computers, we choose to use Raspberry Pis because it has a number of USB ports which we needed to connect different motes, it is relatively cheap and yet powerful, and widely deployed in most IoT projects at the moment. All the Raspberry Pis connect to the server through a LAN switch. They are all assigned IP addresses dynamically by the DHCP service installed on the server as explained in Subsection 5.2.1.

One OpenMote [6], BLE [4], and TelosB mote [8] connect to one Raspberry Pi via USB and this makes up one IoT node as earlier shown in Figure 5.1. The OpenMote and TelosB utilise USB extension cables to reduce congestion on the Raspberry Pi's tightly spaced USB ports. The implementation of the physical network achieves our desired and planned design in Chapter 4.

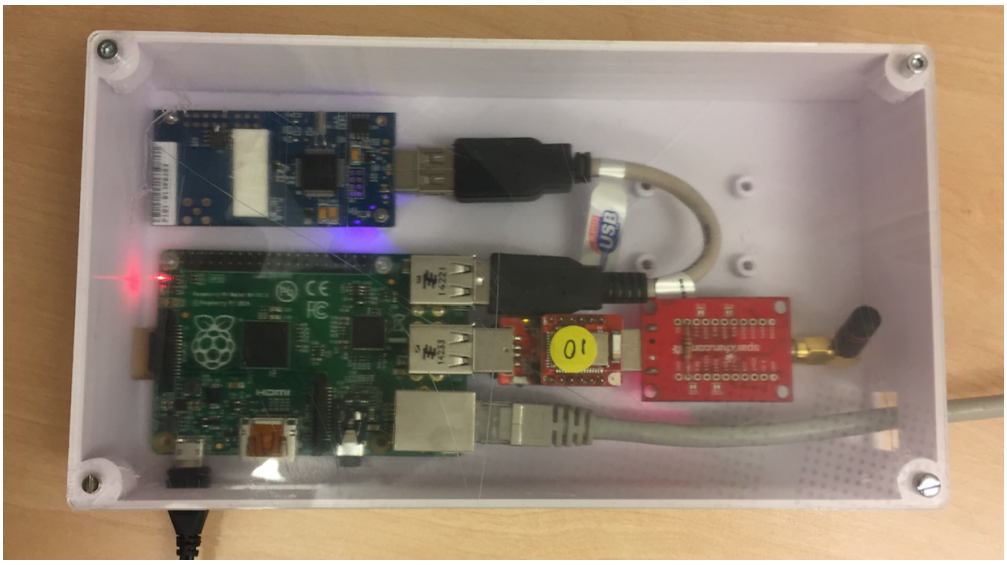


Figure 5.1: A boxed client node with all the motes connected

5.2 Server and back-end

In this section, we discuss technical details about how we have implemented the software components that form the back-end of the testbed. In order for the server and back-end to function properly, we need to install a collection of services. We call the latter the initial server configuration, and we will discuss that first before we proceed to describe implementation details of the software components of the back-end.

5.2.1 Initial server configuration

The client nodes connect, through the Chalmers network, to the server which in turn hosts the back-end of the testbed. In order for the server to work properly in that role, it needs software in the form of services and libraries. First of all, we have installed and configured DHCP-service on the server. DHCP is very useful in configuring client interfaces and gives an administrator complete control over a large network. For a network with many clients, an administrator makes changes in a few DHCP configuration files and restarts the service for changes to take effect in the entire network. This is important so that the client nodes can obtain IP addresses automatically when they connect to the server.

An essential role for the server is to remotely control the client nodes, in connection with running experiments on the client nodes. The server can utilise SSH in order to remotely control the client nodes, but in that case the server needs to connect to each of the client nodes separately. This can work, but it is time consuming and kills any chance of achieving any synchronisation when programming the motes. We therefore make use

of the parallel-SSH (PSSH) library in order to solve this issue. With the mentioned library, an SSH-command can be sent to several client nodes simultaneously.

5.2.2 Database

The back-end of the testbed utilises a MySQL database, which is light-weight and that suits our web-service perfectly. For the choice of database, we also have considered SQLite and PostgreSQL. SQLite has the advantage of not being a client-server engine, rather it is built into the end program. This feature makes SQLite less attractive for us considering our back-end design, where the database is a server and the web-service is a client that contacts it when it needs information. This designs offers more flexibility when adding new components to the back-end, which is not a current issue but can be in future development. We have also considered PostgreSQL, which is more advanced and offers more features than MySQL. The reason that we have not chosen MySQL over PostgreSQL is that we did not have any experience in databases prior to this thesis, and MySQL is simple and offers great amount of community support due its popularity. Furthermore, our utilisation of the database does not go beyond CRUD(Create, Read, Update and Delete). Therefore, even if PosgreSQL is implemented, its advanced features will not be utilised.

The server hosts the database and the web-service connects to it through a port. The database hosts the "memories" of the testbed; the users it has, the experiments it has executed as well as those which await execution. The web-service is the only software component that accesses the database

The information that the database stores, is used by the web-service for 3 main purposes. The first is to perform user authentication of testbed users. The second purpose is to keep track of the user-sessions that are currently open. The third is to help the web-service to determine what parameters to invoke the server testbed scripts with. Other purposes include retrieving experiment logs as well as the testbed job history.

The database contains five tables; *users*, *tokens*, *pending-jobs*, *current-job* and *completed-jobs*. The *users* table contains login information about all the testbed users, as well as other account related information such as the role of the user and last login date among other information. The *tokens* table stores session related information about users that are logged in on the testbed. This table makes it possible for the web service to remember a user after just logging in once. This continues till the user either logs out, or becomes inactive for 30 minutes.

The *current-job* table contains only one entry, and that is the current experiment running on the testbed. The table stores information about the experiments start date, duration and the location of its firmware among other information. The *pending-jobs* table contains all the experiments that the users create and which are successfully scheduled, except the one in the *current-job* table. The information about the experiments

in these two tables are identical, the only difference is that experiment that is currently executing is in the *current-job* table, and the rest is in the *pending-jobs* table.

The last table in the database is *completed-jobs*. This table contains information about experiments that have completed their execution on the testbed. The most interesting information this table stores is the location of the log file(s) of the experiment. If the user would like to get a hold of the data that the mote(s) dumped during the experiment, then the location in the completed-jobs table becomes very useful.

5.2.3 Web-service

All the operations on the testbed that the client requests go through the web-service. We have utilised the Dropwizard framework when building the web-service, as it offers great help when building RESTful web-services. Dropwizard is nothing new in its own right, but rather a collection of Java-libraries for building RESTful web-services, such as Jersey for example. We have chosen Java because we have experience coding in it from earlier projects, as well as its popularity as language generally as well as a back-end-language specifically.

As we have mentioned earlier, the database that we utilise is MySQL, which is lightweight and serves greatly for our purposes. The database consists only of 5 tables and does not have a complex nature, which also makes a MySQL database a suitable choice for the web-service of the testbed.

The web-service utilises the database in order to store all the information about the testbed users and their experiment details. It also keeps track of when users are logged in or not. As we have mentioned earlier, the communication between the client and the server is stateless. Anything that the web-service needs to remember, must therefore be stored in the database. The communication between the web-service and the database is done with the help of a Java-framework called Hibernate, which we utilise to map objects to our relational-database. Object-relational mapping makes the communication between the web-service and the database have an object-oriented nature, and therefore makes the code of that communication similar to the rest of the code for the web-service. This in turn makes the code of the web-service more scalable and reusable. Furthermore, code generated using Hibernate will work with all kinds of databases, while JDBC-API(Java DataBase Connectivity) for example is database-specific. This is a desired feature, as we have mentioned earlier that the implementation of the database might change in future development.

In Figure 4.3, we have discussed how the web-service utilises several scripts in order to communicate with and program the nodes in the testbed. There are 2 different scripts for 2 different purposes, namely start and stop. When the testbed user creates an experiment, the firmware, along other parameters such as the nodes that will be used in the experiment and what platforms for example, is stored in a folder for that

experiment and an entry of the created experiment is added to the table of pending jobs in the database. When the time of the experiment arrives, the script for starting the experiment is run with the path to the experiment folder as a parameter. The script then communicates with nodes that are a part of the experiment, copies the firmware to them and calls the platform-dependent scripts in order to program and start firmware on them.

When the time for running experiment is up, the script for stopping the experiment is run and the script once again contacts the nodes that are a part of the experiment, retrieves the logs that they have obtained and finally cleans up after the completed experiment so that the next one can start. The experiments run non-preemptively, in other words an experiment cannot be paused and continued at a later time.

5.2.3.1 User interface

The testbed users interact with the web-service through a REST API that serves as the testbed's current and only user interface. In order to communicate with the service, a client sends a request to it that follows the specifications of the API. As we have mentioned earlier on, the web-service consists of several resources or components. Depending on what the client aims to perform on the testbed, it sends a request to the resource that corresponds to that operation. The request consists of the URI for that resource as well as the parameters that the resource expects. An example of such a request looks like the following:

curl –data "username=userpassword=psd" localhost:8080/testbed/users/login

The client wishes to login, so he/she sends the above request to the login-resource on the web-service. The URI in the request is *localhost:8080/testbed/users/login*, that is the location of the resource and it helps the web-service to successfully identify the resource that the client is requesting to interact with. In order to login, the client needs to supply a valid username and password combination. In other words, the username and password are the parameters that the login-resource expects to receive so that it can perform its job, which in this case is authenticating the user. In case of successful authentication, the resource returns a session-id which the client needs to send in the subsequent requests that it sends to the web-service. The implementation of a session-id means that the client does not need to authenticate itself every time it needs to access a resource. This feature does not make a big difference when sending requests directly to the web-service, it becomes more important when implementing graphical user interfaces. In such interfaces, the client is asked after authentication only once per session, otherwise the interface becomes less user-friendly. Let us assume that the client wishes to delete a pending experiment, it sends the following request to the deletejob-resource:

curl -i -H "Authorization: bearer 88b32740-e960-4961-9bff-ae0103cd89fa" localhost:8080-/testbed/deletejob/x

The URI in the request is *localhost:8080/testbed/deletejob/x*, where x is the id of the experiment or job that the client wishes to delete. The parameter that the resource expects, experiment id, is a part of the URI. This is an alternative way of providing a parameter value to a resource. The rest of the resources of the testbed include logging out, signing up, creating a job, stopping a job and retrieving the logs of a completed job. All the requests for these resources that client sends follow the same principle as the examples illustrated above. They only differ in the URIs as well as the parameter(s) they contain.

5.3 JTAG Emulator

To implement the JTAG emulator design, we use the Raspberry Pi and the OpenMote CC2538 for the hardware. This is because the Raspberry Pi and the OpenMote CC2538 have the GPIO pins and JTAG interface respectively for establishing JTAG communication. We install a set of Linux dependence libraries and tools on the Raspberry Pi and some of them include:

- WiringPi: As the name suggests, this is a library that makes wiring of the Pi's GPIO pins possible. It is an open-source library written in C that provides a command-line utility "gpio" which is used to control GPIO pins e.g reading and writing to them [32].
- libftdi: An open-source USB driver library for talking to Future Technologies Devices International (FTDI) chips. The latter has manufactured the OpenMote's chip, so the Pi needs the libftdi library to recognise it as a target.
- gdb-arm-none-eabi: This is a GNU Debugger for embedded ARM bare-metal targets. It is installed in order to create a debugging interface for the OpenMote.
- OpenOCD 0.9.0: This version of OpenOCD library has GPIO bit banging capabilities. Bitbanging is a cost-friendly technique of controlling serial communication in embedded systems through software rather than dedicated hardware. As we have mentioned earlier in Subsection 2.2.4 that OpenOCD is a very big open-source community with dedicated contributors, the library consists of tens of configuration files for different boards from different vendors. A configuration file for OpenMote CC2538 is also included in this library and we modify and utilize it for our interests. We also modify and utilise a configuration file for BCM2835 chip which is used by the Raspberry Pi.

We set each of the JTAG lines TDI, TDO, TMS, TCK, RST, and GND from the OpenMote to their respective GPIO pins on the Raspberry Pi. The Appendix Section A of this report, contains all the detailed wiring connections of this JTAG emulator. We use ARM Cortex 10-to-20 pin JTAG adapter and male to female jumper wires to connect the OpenMote's JTAG lines to specific GPIO pins on the Raspberry Pi as shown in Figure 5.2. This connection is supplemented with installing, on the Raspberry Pi, the

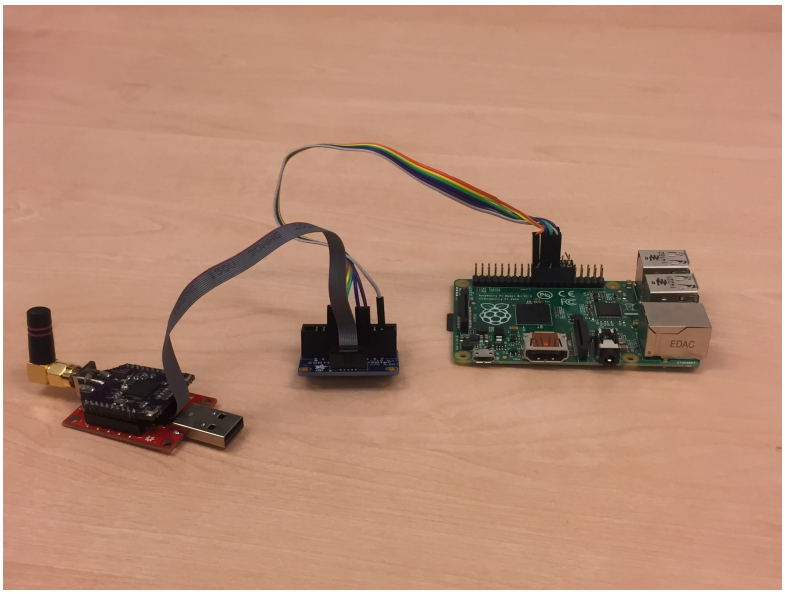


Figure 5.2: Raspberry Pi JTAG emulator setup showing cable connections from the OpenMote's JTAG header to the Raspberry Pi's GPIO pins. The mote gets powered from the Pi's USB port.

dependence libraries and tools listed above. Running a script on the Raspberry Pi which contains OpenOCD configuration commands starts OpenOCD (gdb server) on the Pi. Port number 3333 is the default OpenOCD port.

Launching $arm - none - eabi - gdb$ in another terminal and issuing the gdb command *target extended-remote <ip address of the Raspberry Pi:3333 >*, starts a gdb client which communicates with the target (OpenMote) on the gdb server via TCP port number *3333*. A user now has full control of the OpenMote through JTAG interface and can perform both hardware and software debugging of the mote as well as programming it.

A user who is debugging on the OpenMote does not require physical access to the Raspberry Pi on which the mote is connected. On the testbed server, we have implemented port forwarding through SSH. This is also known as SSH tunnelling [33] and it creates a secure connection between a local computer and a remote machine. The remote machine in this case is the Raspberry Pi which has the OpenMote connected to it whereas the local computer is the user's computer. The user will only need to be authenticated to the testbed's server and it is the server which will tunnel his/her connection to the Raspberry Pi computer. He/she does not have any knowledge about the remote machine's IP address details and this is why we chose SSH tunnelling to be a good implementation because of the secure environment that it provides. Instead of using the Pi's IP address and the default port 3333 mentioned above while connecting to the OpenOCD server, a user just issues the command *target extended-remote localhost:xxxx* from his or her local machine after authenticating to the server. *xxxx* is the dedicated JTAG debugging port

configured in the testbed's server.

For this testbed, JTAG programming and debugging are set up on only two nodes for demonstration purposes. The rest of the nodes' motes are programmed through USB interface (for the BLE) and the BSL script [34] (for telosB motes and the remaining OpenMotes). Even though the implementation of our JTAG design is cheap in terms of costs, there are other bottlenecks which we face along the way during the implementation phase. The connection from the OpenMote's JTAG interface to the Raspberry Pi's GPIO pins as shown in Figure 4.7 is delicate. Possibilities for short-circuits to occur to both the Raspberry Pi and the OpenMote are high in case 5V are accidentally supplied to the board instead of 3.3V. This can instantly blow up the Raspberry Pi and the mote and render them useless. Another challenge was getting the right types of cables and wires to connect the OpenMote's JTAG interface to the Raspberry Pi. The OpenMote has a small 10 pin JTAG header and getting the right cable that fits in that header took us considerable time as we waited for its delivery from the online store supplier. Another challenge in implementing the JTAG emulator design is that many libraries and drivers that enable full communication between the OpenMote and the Raspberry Pi were deprecated and yet the new drivers were not yet compatible with the Raspberry Pi's operating system. This took us much time to figure out and solve.

5.3.1 Programming of other motes

The JTAG emulator is designed and implemented to exclusively work on sensor motes and other micro-controller boards which have JTAG pin interfaces. A majority of the sensor motes and micro-controller boards without JTAG pin interfaces can always be programmed via USB. For our case in this thesis, both the OpenMote and the telosB have support for JTAG but we have chosen to implement JTAG debugging on the former mote because it has better performance specifications than the latter.

The majority of the OpenMotes in our testbed are however not going to be programmed via JTAG. We choose to use a BootStrap Loader (BSL) script [34] to program the rest of the motes because it is simple and time-saving. One downside of using the BSL script is that it does not perform debugging. Another limitation of the BSL script is that it cannot reprogram a mote which had been previously programmed with the bootloader backdoor disabled.

6

Evaluation & Discussion

In this chapter, we evaluate the three major features of our testbed namely JTAG debugging, logging, and scheduling. We also evaluate memory management in the testbed and CPU utilization of the nodes. We present the findings in graphs and tables with explanations. Upon evaluating each feature, we discuss the obtained results and draw conclusions. In the last section of this chapter, we summarise all the evaluation results. We close the chapter with a table showing all the features and fuctionalities that our testbed has and where each of them was picked from. The sources are the three testbeds where we drew our inspiration from at the start of this thesis.

6.1 JTAG debugging

In this section, we describe the case study that we carried out for JTAG programming and debugging with our implemented JTAG emulator tool. We go on to evaluate its functionality, performance, and the advantage that this tool provides to embedded system programmers who use it compared to those that use other methods.

6.1.1 Case study

We have implemented our inexpensive JTAG debug module from the Raspberry Pi's GPIO pins, OpenMote cc2538, simple jumper wires, ARM cortex adapter and the OpenOCD [15] library as explained and illustrated in the implementation part of this report (Section 5.3). In this section, we evaluate the functionality and performance of our JTAG debug module against the existing standard JTAG debug modules such as the Jlink Segger [35]. We check to see if our debug module can perform the same tasks as its commercial counterparts (functionality) and how accurate is it compared to the commercial module (performance). Below is how we check and determine the basic functionality of our JTAG debug module.

(a) Connecting to the sensor mote and interacting directly with its CPU. We are able to halt, reset and resume the running of the OpenMote cc2538 from a remote computer using our JTAG debug module via a GDB client terminal.

(b) Loading and flashing of firmware onto the mote. With our JTAG debug module, we are able to successfully load and flash firmware directly onto the mote without worrying about bootloader backdoor problems. For example in Figure 6.1, we connect to the mote, halt its CPU and load the firmware onto it (*hello-world.elf* Contiki program). Any other firmware can therefore be successfully programmed onto the mote using our JTAG emulator just as it would be done with any other commercially available JTAG programmers.

(c) Debugging programs running on the sensor mote. With our JTAG debug module, we are able to set breakpoints at any location of the program which we are debugging. The location can either be a memory address, line number or at the beginning of a particular function name. With breakpoints set, we are able to step through the program, analysing it on a line by line basis. With the use of GDB commands, we have full control of the program's flow and behaviour through our JTAG programmer. For example we are able to track which part of the program is currently executing, see the lines of code in that particular section of code, step forward and backward in the code, list all the set breakpoints and view information about them, among other features.

6.1.2 Evaluation of our JTAG module when programming and debugging the OpenMote

We compile a simple *hello-world* contiki program and use it specifically to evaluate how our JTAG debug module works when debugging a program. We generate an executable object file with debugging information in it and load it in gdb (*hello-world.elf*). We are using *arm-none-eabi-gdb* tool chain for ARM architecture because it is the only one which is compatible with the OpenMote.

We are able to load the object file into gdb client after invoking a JTAG connection to the OpenMote with OpenOCD (which acts as the gdb server). We check if any breakpoints are set. There are none initially. We then set breakpoints on lines 49 and 56 in the *hello-world.c* file as shown in Figure 6.1. The hello-world.elf object file contains all the contiki source code files but in this case we are only interested in the *hello-world.c* file for demonstration purposes. That is why we specify the filename while setting breakpoints. We then run the program and when the first breakpoint is hit, the program temporarily pauses and we can examine the section of code which has the breakpoint. This is shown in Figure 6.2.
We are able to step through the program a few lines upfront or behind and do back-tracing. Back-traces show a summary of how the program is running and the steps it took to reach where it is. This allows the programmer to examine all the frames on

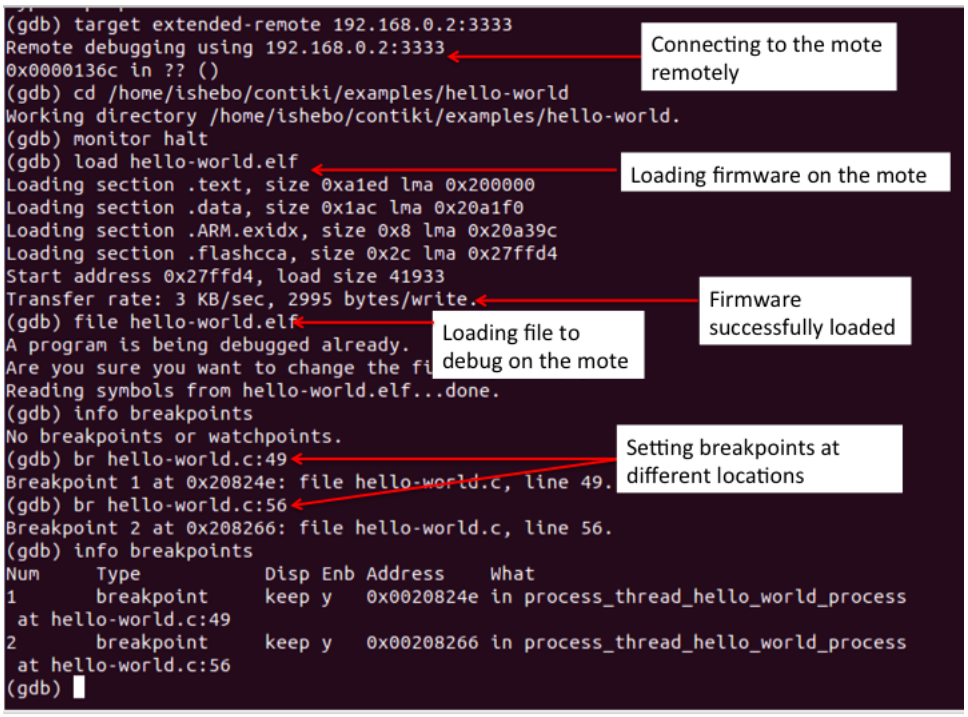


Figure 6.1: Programming and control of the mote via our JTAG emulator. We temporarily halt the CPU of the mote, load the firmware on it and chose a program file to debug. We set break points at different lines and later show all the set breakpoints.

a line by line basis. We can also check values of variables in the program and even temporarily assign new values while the program is still running. This is important in debugging when a programmer needs to check with different values. We can as well set watchpoints, check memory registers and perform other debugging tasks as we would do with the standard commercial JTAG programmers. We can safely say that our JTAG debugging feature works as it is supposed to.

However, once in a while during a remote JTAG debugging session on the OpenMote, a gdb client connection to the gdb server (OpenOCD) gets disrupted and loses the connection to the server. Also, on rare occasions the server panics upon initialisation. We base on such occasional failures to determine the performance of our tool and we unbiassedly rate its performance at 70% when compared to the commercial JTAG debug module. The remedy to the two failure issues we have mentioned above is to retry and reconnect. We have implemented a script to retry launching automatically upon first failure.

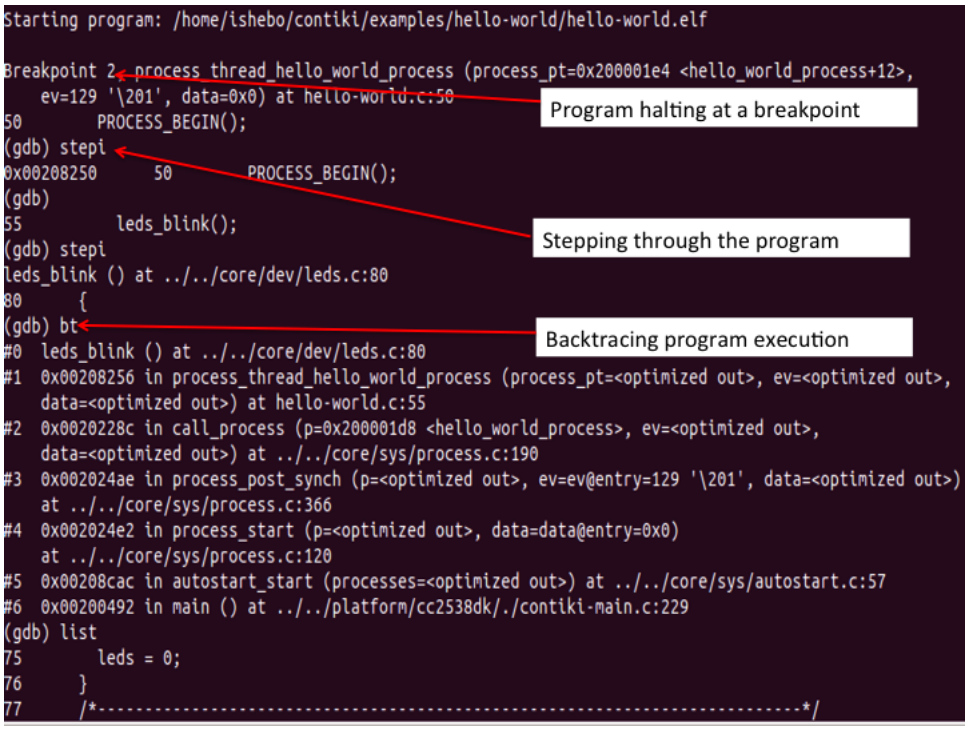


Figure 6.2: Figure showing breakpoints enabled, temporarily suspension of program execution at line 50, and monitoring of the stack frames with the backtrace command.

6.2 Logging

In this section, we will evaluate and discuss the logging of data in the testbed. We will consider two aspects of logging, namely performance as well as the maximum logging rate.

6.2.1 Logging performance

The first aspect of logging that we will consider is its performance against increasing the number of nodes or Pis, that are involved in the experiment and are logging simultaneously. In order to evaluate the performance, we have run several experiments on the testbed, where each experiment has the duration of one minute. The number of nodes that log data in the experiments differs however.

We obtain the results for the evaluation by first performing two experiments with only one node, and recording the number of log-lines captured and written successfully by the node. Thereafter, we increase the number of nodes by one and perform two experiments again as well as recording the number of log-lines and so on up to four nodes. In order to save time, we thereafter only run the experiment with 20 nodes in order to make sure that the behaviour is the same. We plot the number of log-lines obtained in the

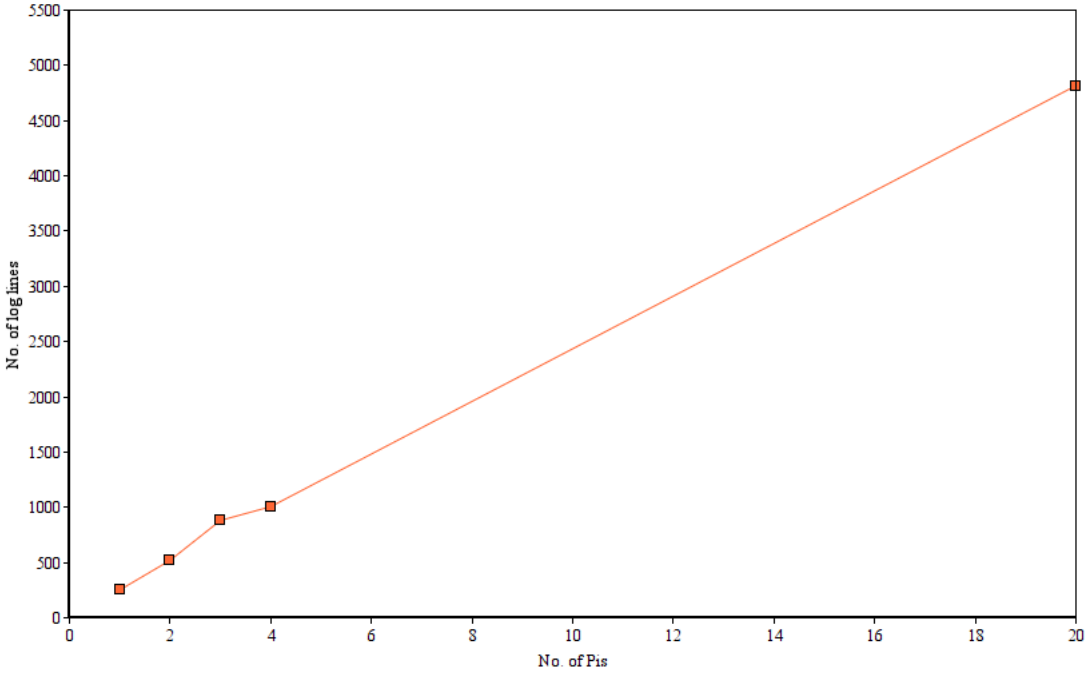


Figure 6.3: Number of log-lines totally received in proportion to the number of Pis involved in the experiment

experiments against the number of nodes, or Pis, that were involved in them and we obtain the plot as shown in Figure 6.3.

The figure yields a linear behaviour, which is what is expected considering the nature of the experiments. If one node produces x number of log-lines, then two nodes are expected to produce 2x log-lines, three nodes are expected to produce 3x log-lines and so on. In an ideal fault-free environment, that would be the case. Unfortunately, that does not apply for our testbed setting, where there are several factors that can affect the number of log-lines obtained from the nodes. For example when programming the nodes, certain time is lost between when the node receives the instructions to start and when it actually starts. This in turn depends on what is running on the node itself, for example other programs and services. Despite this, the behaviour obtained from the recorded data lean towards a very straight linear behaviour.

6.2.2 Maximum logging rate

The second evaluation that we will consider for the logging is the maximum logging rate. This evaluation aims to find the limit for where logging is obtained in its entirety and no log-lines are lost. This limit-value is important, both for us the developers as well as for future testbed users, as it reveals the maximum speed in which the nodes can log data. Experiments on the testbed should therefore avoid going beyond that limit, as log data will be lost.

The evaluation process consists of performing several experiments on one node. The duration of these experiments is the same, one minute each, and the baudrate is set to 460,800 in all of them. In each experiment, the node runs a program that writes one line to the serial port per time unit. Each line contains a counter that increases per line, which indicates the number of that line. This is added in order to keep track of the recorded lines and discover any missing ones. The only difference between the experiments is the time unit in which a new log-line is written to the serial port.

In order to obtain the maximum logging rate for the testbed, we increment the time unit gradually. With every increment of this value, a new experiment is performed and the number of log-lines that are lost during the experiment is calculated. We plot the recorded results in Figure 6.4. From the figure, we notice that the maximum logging rate is around one line per 0.007 milliseconds. Logging beyond that rate causes the node to loose a number of the log-lines. This number increases as the frequency increases. This rate is the same regardless of the number of Pis in the experiment.

6.3 Memory and CPU usage

In this section, we evaluate two other significant aspects of the testbed, namely memory management and CPU usage. Examining these two factors allows us to determine the maximum upper limit to which we can push our testbed without degrading its services.

6.3.1 Memory management

One of the questions that needs to be addressed is how much space can an experiment maximally take. This information is critical to find in order to estimate the appropriate amount of storage that should be provisioned at the server side to cater for testbed experiments. An experiment consists of an entry in the database, firmware(s) and experiment related text files as well as logging. The space for the entry in the database is negligible. The experiment related text files can be neglected as well because these files contain small amounts of data and their size is therefore negligible. The size of the firmwares uploaded by the user as well as the log files created by the testbed on the other hand are not negligible, and are therefore taken into consideration when estimating the amount of space that an experiment can take.

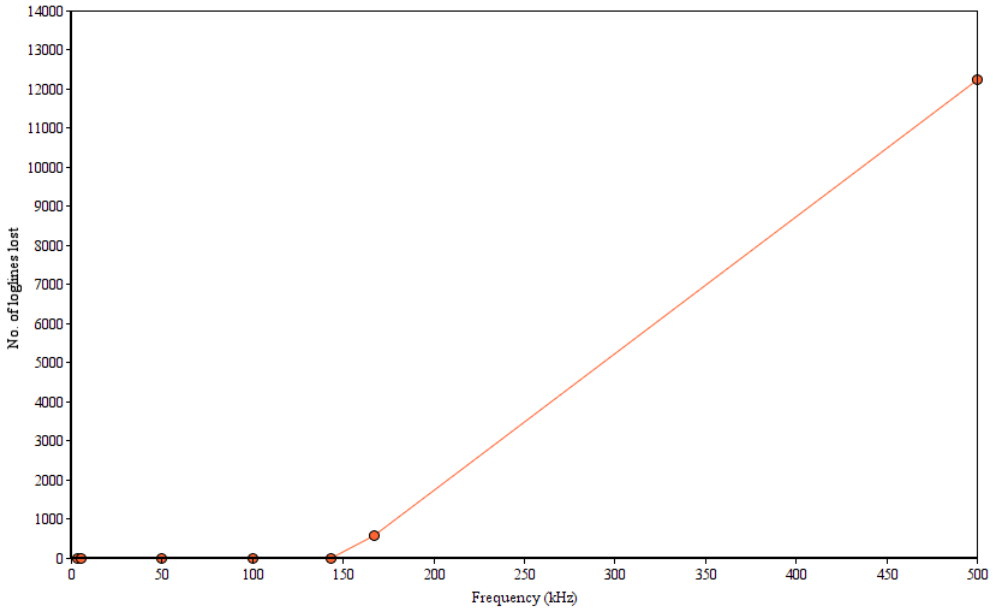


Figure 6.4: Logging frequency and the amount of log-lines lost. When the frequency increases, a greater amount of log-lines is lost

The size of the log files the testbed creates for the experiments depends mainly on two factors, namely the logging rate and the number of nodes involved in the experiment. When a firmware running on a node is logging at the maximum rate, every second of execution measures up to five kilobytes of space for logs on the server. This value was calculated by running an experiment on a node for 110 seconds, and thereafter dividing the size of the log files generated on the server by the number of seconds. This experiment was repeated several times and an average value was taken. See Table 6.1 to check for how we calculate the total amount of space an experiment takes.

An experiment that logs at the maximum rate and utilises all the 25 nodes will generate 25*5 = 125 kB/s. If this experiment runs for the maximum amount of time, which is 24 hours, then it generates 125*24*60*60 kB $\approx$ 10.8 gigabytes. This value implies that the server needs to allocate a very large amount of storage space (in terabytes) in order to be able to store the logs of multiple experiments running on multiple nodes at maximum rate.

For the nodes, an experiment generates almost 3 kB per second on each node. This translates to $\approx$ 253 megabytes. This means that on each node, at least 253 megabytes

	Number of uploaded firmware: x	**Logging received from the Pis**	**Total**
Space allocated in kB	The total size in kB for the x firmwares together	(Number of nodes utilised in the experiment: y, experiment time: z seconds) At max logging will take (y*z*5) kB in space	total space needed = total size of firmwares (kB) + y * z * 5 kB where y is the number of nodes utilised in the experiment and z is the running time of the experiment in seconds)

Table 6.1: Space allocation for experiments on the server. The amount of logging received from the Pis and the size of the firmwares that the user uploads are the two main factors for the amount of space an experiment can take

plus the firmware size, must be available in order to accommodate with an experiment that logs at maximum capacity and runs for the maximum of one day. This does not cause any problems to the Pis, as they have much more space available on them than the 253 megabytes.

6.3.2 CPU usage

We measure the CPU utilisation of the node particularly the Raspberry Pi when experiments are running. CPU utilization is measured and recorded under three situations as shown in Figure 6.5. It is at 2.3% when there is no job running on the Pi node, 7.7% only when a BLE job is running, and 13.5% when both JTAG debugging and a BLE job are running at the same time.

In general, total CPU utilisation of the nodes in our testbed is below 40% when we are running our experiments and we can safely say that the node's CPU is underutilized at this stage. This means that testbed users have enough room and resources on the nodes to plan and carry out other types of experiments without straining the testbed. However, excessive deployment of experiments on the node may cause deteriorated performance of the testbed. For best results, we recommend that CPU utilization of one node should not exceed 85% when the user's experiments are all running on the node at the same time.

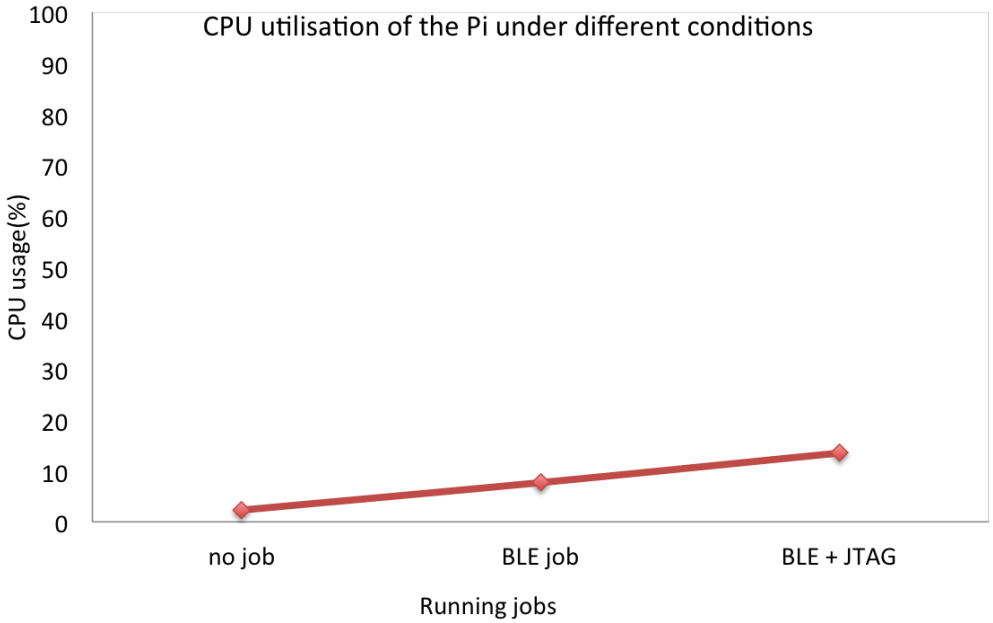


Figure 6.5: This figure shows the varying CPU utilization of the Pi when it is idle, when only a BLE job is running, and when both BLE and JTAG are running. The graph shows that there are ample CPU resources for users to run other experiments.

6.4 Scheduling

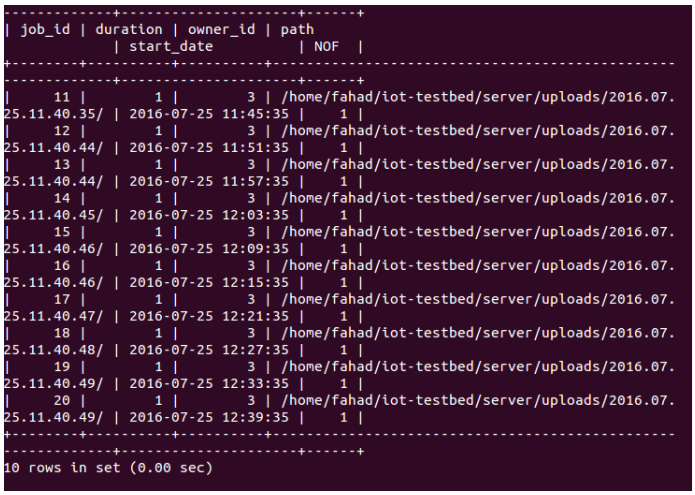
For the web-service, we have implemented a scheduler that accommodates the simple scheduling needs of the testbed. The scheduler does not rely on a specific scheduling algorithm, that is why we have chosen to evaluate it to make sure that it behaves as expected.

In order to test the scheduler, we consider several cases and see if the scheduler manages them properly. We of course cannot test every possible case, but the cases that we test cover a wide spectrum, and that is more than enough for its current utilisation in the web-service of the testbed.

When a user aims to schedule an experiment, he/she specifies one of two options, namely asap, and date. The former schedules an experiment as soon as possible, and the latter schedules an experiment at an exact date and time.

Before proceeding to the different cases, it is worth mentioning that a five minutes buffer-time is reserved both before and after each experiment regardless of the duration of the experiment itself. Furthermore, the duration of each experiment is one minute unless stated otherwise, and the terms job and experiment will be used interchangeably.

For the first case, we start with an empty schedule. We schedule 10 jobs with the asap



```
+------------+--------------------+------+
| job_id | duration | owner_id | path
           | start_date          | NOF  |
+--------+----------+----------+-----------------------------------------------
------------+--------------------+------+
|     11 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.35/ | 2016-07-25 11:45:35 |    1 |
|     12 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.44/ | 2016-07-25 11:51:35 |    1 |
|     13 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.44/ | 2016-07-25 11:57:35 |    1 |
|     14 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.45/ | 2016-07-25 12:03:35 |    1 |
|     15 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.46/ | 2016-07-25 12:09:35 |    1 |
|     16 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.46/ | 2016-07-25 12:15:35 |    1 |
|     17 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.47/ | 2016-07-25 12:21:35 |    1 |
|     18 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.48/ | 2016-07-25 12:27:35 |    1 |
|     19 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.49/ | 2016-07-25 12:33:35 |    1 |
|     20 |        1 |        3 | /home/fahad/iot-testbed/server/uploads/2016.07.
25.11.40.49/ | 2016-07-25 12:39:35 |    1 |
+--------+----------+----------+-----------------------------------------------
------------+--------------------+------+
10 rows in set (0.00 sec)
```

Figure 6.6: First case where 10 jobs, with the duration of one minute, are scheduled with the asap option. The actual time between jobs is 6 minutes due to the 5 minute buffer-time. For the last experiment scheduled, only the buffer-time before the experiment is visible

alternative. We expected the scheduler to put the first experiment five minutes from the start-time, and then six minutes after that the second experiment has its start-time (one minute for the duration of the first experiment plus five minutes buffer after the experiment finishes), and so on. For 10 experiments, the total buffer-time is 50 minutes and the total duration is nine minutes. The reason why it is nine minutes is because for the last experiment on the schedule, we see only the start time and its duration has no effect on the schedule because there is no experiment scheduled after it.

After we have obtained the theoretical values for this case, we proceed to compare it with the actual values that the scheduler has given the experiments, which are shown Figure 6.6. From the figure we see that the start-time is 11:40:35 and the last job is scheduled at 12:39:35. The difference between these times is 59 minutes as expected.

Now we build further on the first case, by making an attempt to schedule an experiment at a certain time within the time that the asap schedule spans over. The expected outcome of this new case is that nothing will schedule, as the asap schedule is as tight as possible with no time to schedule anything between experiments.

The actual outcome of this case came as expected. Making an attempt to schedule an experiment at 12:00:00 fails, as shown in Figure 6.7. Furthermore, scheduling a job at 12:45:34 fails as well. On the other hand, scheduling a job at 12:45:36 succeeds. This serves as an example of how tight the asap schedule, and how accurate the scheduler is when calculating intervals between the experiments in order to decide if an experiment can fit in that interval or not.

The last case that we evaluate is checking for what happens when we schedule an experiment at a certain time, where at least one experiment can fit before it, and then schedule

```
fahad@fahad-VirtualBox:~/iot-testbed/server/examples/ble-blinkandlog$ curl -i -H
 "Authorization: bearer 88b32740-e960-4961-9bff-ae0103cd89fa" -F "firmware=@outp
ut.hex" "localhost:8080/testbed/createjob/2016-07-25%2012:00:00/1?platform=ble&h
ost=pi1"
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Date: Mon, 25 Jul 2016 09:42:06 GMT
Content-Type: text/plain
Transfer-Encoding: chunked

The job cannot be scheduled at the requested date and time
```

Figure 6.7: Second case where an attempt to schedule a job within an interval that is occupied is rejected by the scheduler.

several experiments with the asap alternative. The expected outcome is that some of the experiments will schedule before the already scheduled experiment, and some after it. The experiments will be scheduled in such a way that the schedule will be as tight as possible.

We start by scheduling a job at 14:20:00, with the duration of one minute. Thereafter at the current time, which is 13:35:17, we schedule 10 jobs with the asap alternative, with duration of five minutes each. The scheduler outputs the schedule shown in Figure 6.8.

The schedule behaves as expected. The first three asap jobs (36, 37, 38) schedule before the scheduled job (35) because there is enough time for them but the fourth asap job (39) schedules after. This is due to the fact that Job 38 starts at 14:00:17, and its duration is five minutes. This means it will end at 14:05:17. Then there is a five minute buffer-time till the next job; which means that the job after that cannot start till 14:10:17. If job 39 would start at 14:10:17, then it would end at 14:15:17 which would leave less than five minutes to the job already scheduled at 14:20:00. This is why job 39 starts after job 35 and not after job 38. For all the cases that we have evaluated, the outcome of the scheduler has been as expected. We therefore conclude that the scheduler behaves as expected.

6.5 Evaluation summary

The evaluation results have come as we anticipated. Our JTAG debugging tool using a Raspberry Pi offers the same functionalities as other JTAG debugging tools, such as the Jlink Segger programmer. The logging performance from the testbed nodes is steady and not affected by the number of nodes involved in a an experiment. The logging performance suffers only when logging is done in higher frequencies, more than 143 kHz to be exact. Logging beyond that frequency causes log-lines to be lost and therefore data will not be able to be captured in its entirety.

An experiment can generate maximally 10.8 gigabytes amount of log-data, which will be stored in the server. This implies that the server must have an amount of storage in terabytes in order to accommodate with the possibility that several of such experiments are run on the testbed. The CPU utilisation on the testbed nodes shows no more

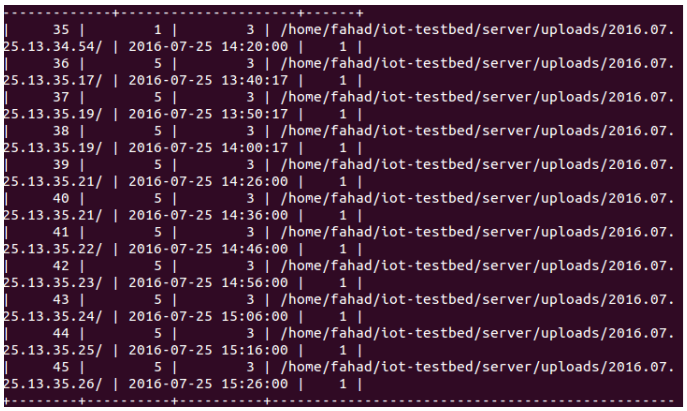


Figure 6.8: Third case where a job, 35, is scheduled first and thereafter 10 jobs are scheduled with asap option. Jobs 36,37 and 38 are scheduled before and the rest is scheduled after.

than 40% utilisation, which implies that testbed users have room and space to plan and carry out other types of experiments without straining the testbed. On the other hand, we have noticed that excessive deployment of experiments on the node may cause deteriorated performance of the testbed. Finally, the scheduling of experiments on the testbed functions as we expected; the scheduler puts the experiment(s) that the testbed users aim to run in a tight schedule that meets the users' scheduling constraints.

6.5.1 Testbed features summary

In Table 6.2 below, we show a set of features which we have in our testbed and where we picked them from. This is intended to affirm one of our main aims at the beginning of this thesis which was to consolidate a number of features from different testbeds into one testbed. We have not evaluated our testbed in comparison to the other testbeds, as this has not been the goal of this thesis.

Feature	**Source**
Scheduling	Indriya, FlockLab
JTAG debugging	Self-inspired
Heterogeneous support	Indriya, FlockLab, TWIST
Reusability	Indriya, Twist
Simplicity	Self-inspired
Automation	Indriya, Twist, FlockLab

Table 6.2: Table showing our consolidated testbed features from the 3 different testbed projects

7

Conclusion and Future Work

In this final chapter, we put forward our concluding remarks about the thesis work and go on to make suggestions about what could be improved in the future work.

7.1 Conclusion

In this thesis , we manage to combine, under one testbed, most of the different services and functionalities which previous IoT and WSN testbeds have offered to users. We extensively review the literature from few of the major testbeds which have been developed. We select the most crucial features from each of the reviewed IoT and WSN testbeds and consolidate them in our testbed. In short, we are not reinventing the wheel but with a combination of inexpensive hardware, and open source software, we design and implement a testbed that offers simplicity, reusability and heterogeneous hardware support. Another contribution of our work is the automation of experiments through the utilisation of a REST API, and JTAG debugging/programming using a DIY tool.

We evaluate features and functionalities and we discuss the results from the evaluation and summarise them in visual graphs. We evaluate the logging functionality, both in terms of speed and scalability, and show that logging rate is not affected by the number of nodes that are involved in an experiment and that the maximum logging rate without losing data is approximately 140 kHz. We evaluate CPU usage, and show that the CPU utilisation on the nodes is well under 40% when we run experiments, which means testbed users have more than 50% of the CPU on each node to utilise in their experiments. We also evaluate memory management, and show that theoretically an experiment can take as much as 10.8 gigabytes on the server just to store all the logs from the nodes. This implies that the server must utilise a form of storage solution that can accommodate this amount of data produced by user experiments. Finally, we evaluate the JTAG emulator tool and the experiment scheduler through case studies, which both show that these tools are working as they are supposed to.

We are very optimistic that our work will be valuable to the IoT and WSN community. The simplicity and reusability of our testbed makes it easy for other researchers to replicate it and thus have a testbed of their own. This will provide researchers with the possibility of not only utilising the testbed to conduct experiments locally but also to develop it further to accommodate for their research needs. In the next section, we recommend what we think should be done in future to improve this IoT and WSN testbed and further the research work in the IoT and WSN sector.

7.2 Future Work

In our work, we did not enforce any security mechanisms in the testbed. With increased connectivity comes increased attack surfaces. In an IoT and WSN ecosystem, it is very important to ensure that the communication among the nodes is secure from external attackers and adversaries. We strongly suggest that future work should be done in securing the communication inside the testbed.

In a distributive computing environment such as IoT's, the node's time synchronisation should be taken into consideration. We did not implement any time synchronisation protocols because this was not in our scope. Our main focus was putting up a testbed which would act as a foundation stone for all these other important requirements. Computers' clocks always drift. Even if this clock drifting can be very small, it can accumulate over a period of time hence resulting into significant errors. Lack of time synchronisation in an IoT testbed has significant effects on the log file accuracy, scheduled testbed operations, auditing, and monitoring of the testbed activities. We therefore recommend that time synchronisation should be added on the list of future work.

Throughout our work, we have only used wired network connections to establish communication across the testbed. This has come with a disadvantage of bulkiness and congestion. We suggest that other networking means such as WLAN and RFID be used in future to increase the mobility of the testbed.

We have, due to time constraints, not implemented a graphical user interface where the testbed user can interact with web-service through a web page or even an application. Currently, the only method of interaction with the web-service is through sending direct requests that follow the specification of the API. This works fine for users aiming to write scripts to interact with the testbed, but for users that wish to interact through a GUI, it is not possible at the moment. The foundation for building any interface on the other hand, has already been laid down. The API is generic and can be used from any platform in building any GUI. This will save enormous amount of time as the only work that is left is building the interface on the client side and connecting it to the API, and everything else is already complete.

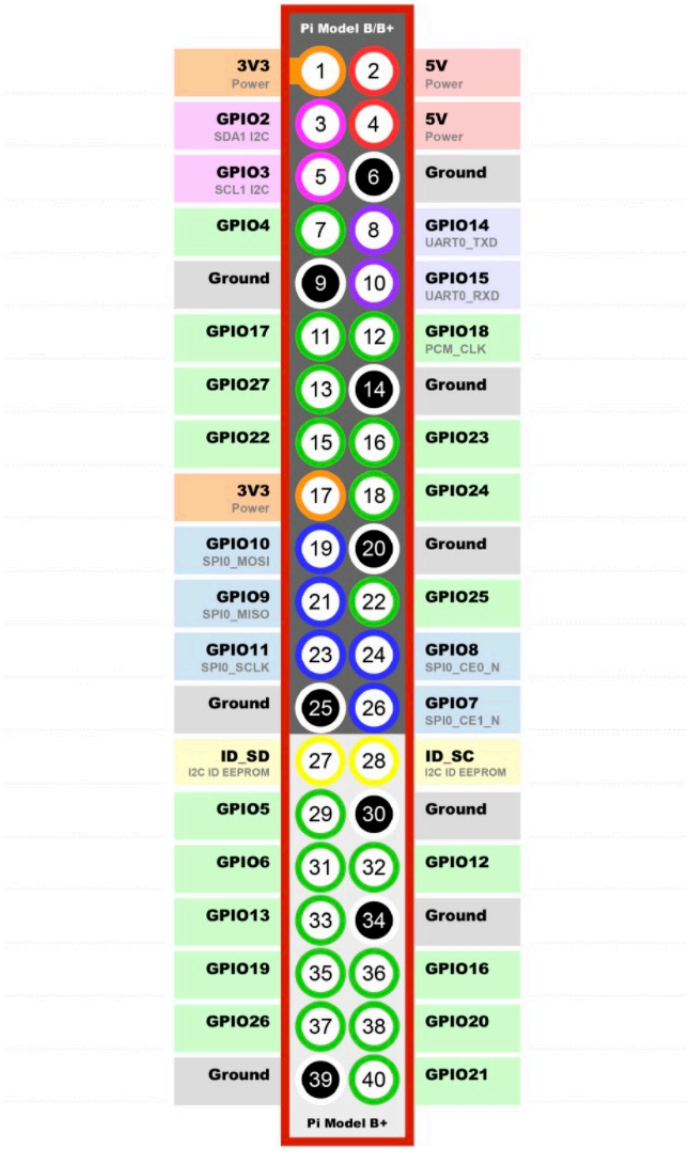
Appendices

A

JTAG debugging

In the Appendix sections below are the pinouts and schematics of the Raspberry Pi's GPIO, the 10-pin to 20-pin JTAG adapter, and the OpenMote CC2538. The focus is on the JTAG lines and which GPIO pins they connect to. The final connection mappings are shown in Appendix A.4

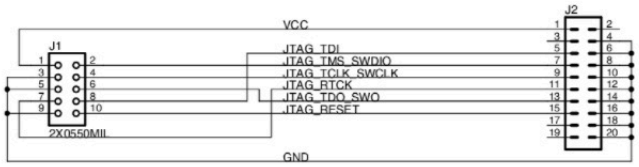
A.1 RPi Model B+ GPIO pinout



A.2 10-to-20 pin JTAG adapter

10 pos (50 mil pitch) connector

20 pos (100 mil pitch) connector



ARM 10-pin interface Serial Wire Mode

1-VCC	2-SWDIO
3-GND	4-SWCLK
5-GND	6-SWO
7-N/U	8-N/U
9-GND	10-RESET

ARM 20-pin interface Serial Wire Mode

1-VCC (Vtref)	2-N/C
3-N/U	4-GND
5-N/U	6-GND
7-SWDIO	8-GND
9-SWCLK	10-GND
11-N/U	12-GND
13-SWO	14-GND
15-RESET	16-GND
17-N/C	18-GND
19-N/C	20-GND

ARM 10-pin interface JTAG Mode

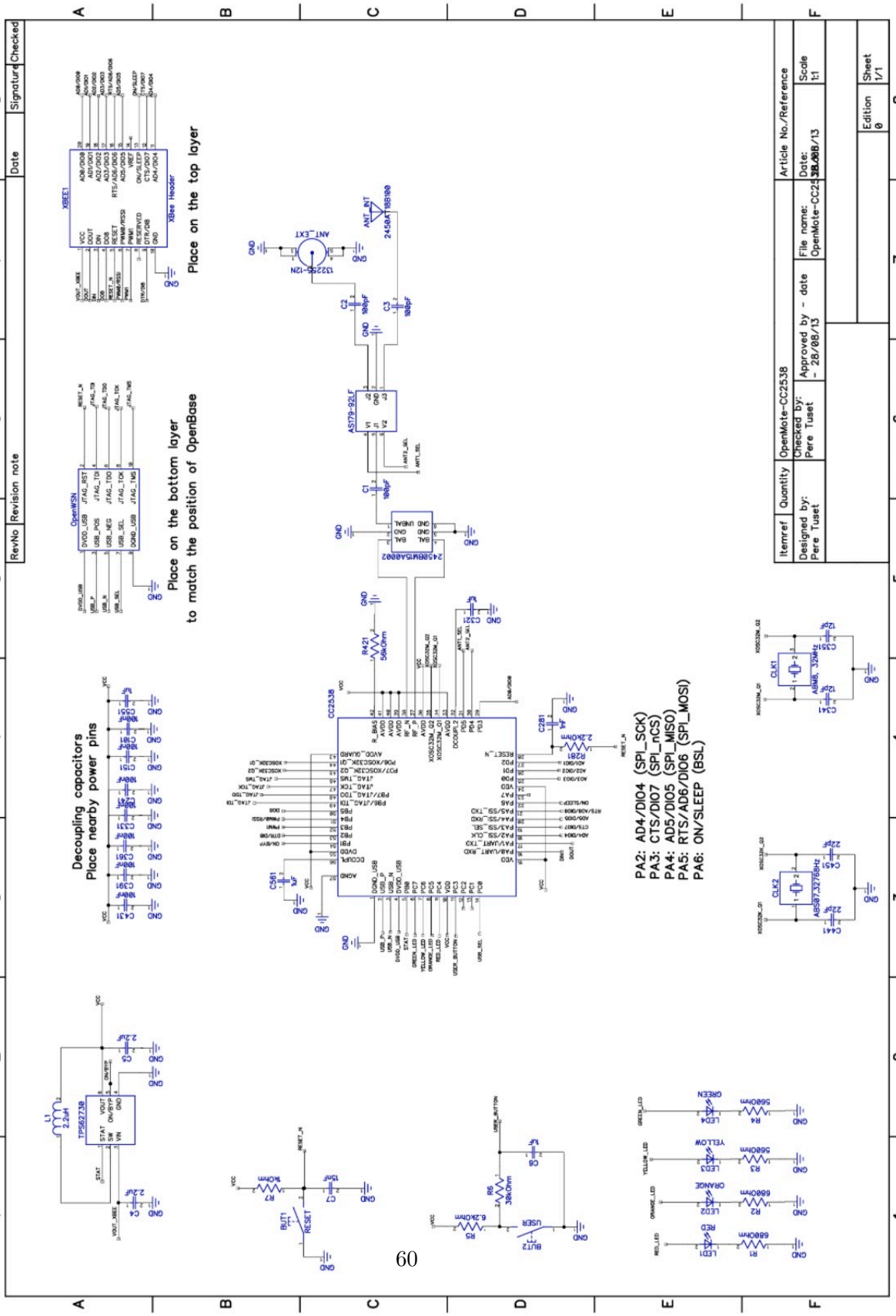
1-VCC	2-TMS
3-GND	4-TCLK
5-GND	6-TDO
7-RTCK	8-TDI
9-GND	10-RESET

ARM 20-pin interface JTAG Mode

1-VCC (Vtref)	2-N/C
3-N/C (TRST)	4-GND
5-TDI	6-GND
7-TMS	8-GND
9-TCLK	10-GND
11-RTCK	12-GND
13-TDO	14-GND
15-RESET	16-GND
17-N/C	18-GND
19-N/C	20-GND

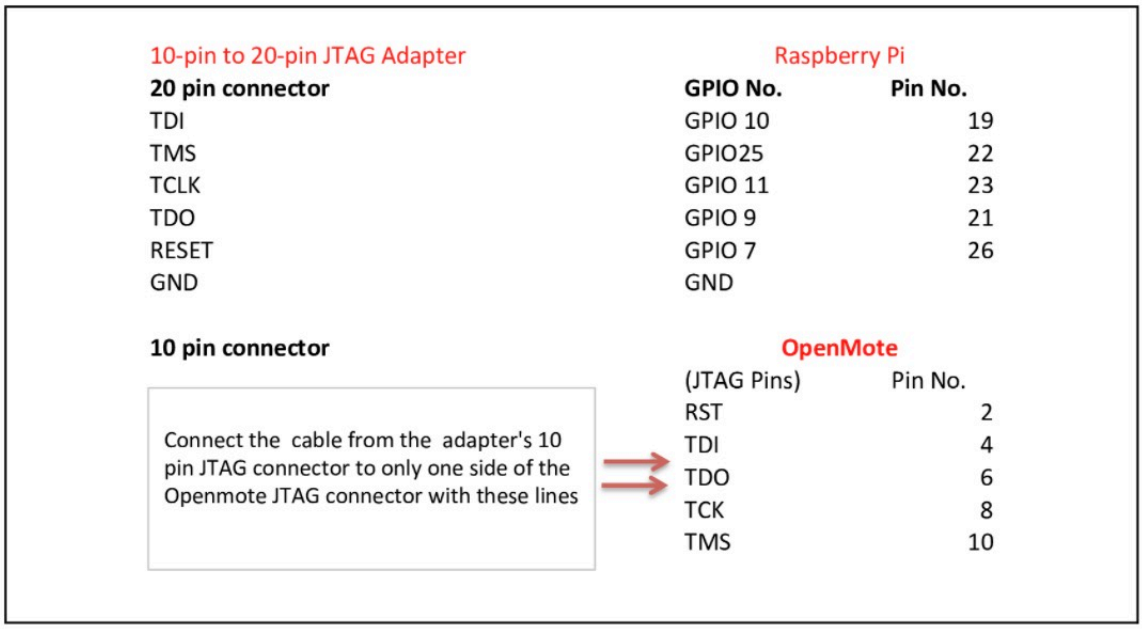
A.3 Openmote schematic

The area of interest in this OpenMote CC2538 schematic is the one labelled OpenWSN with JTAG pins 2, 4, 6, 8, and 10.


Place on the top layer
Place on the bottom layer
to match the position of OpenBase
Decoupling capacitors
Place nearby power pins
PA2: AD4/DIO4 (SPI_SCK)
PA3: CTS/DIO7 (SPI_nCS)
PA4: AD5/DIO5 (SPI_MISO)
PA5: RTS/AD6/DIO6 (SPI_MOSI)
PA6: ON/SLEEP (BSL)
OpenMote-CC2538
Designed by: Pere Tuset
Checked by: Pere Tuset
Approved by - date - 28/08/13
File name: OpenMote-CC2538
Date: 28/08/13
Scale 1:1
Sheet 1/1
Edition 0
Article No./Reference
Quantity
Item/ref
RevNo
Revision note
Date
Signature
Checked

A.4 Overall connection

This is how the connections from the Raspberry Pi's GPIO pins, the 10-pin to 20-pin JTAG adapter, and the OpenMote JTAG connector look like after connecting everything together.



Bibliography

[1] "The Internet of Things: How the Next Evolution of the Internet Is Changing Everything," White Paper, Cisco, Apr. 2011.

[2] Ericsson AB, "Ericsson Mobility Report: On The Pulse of The Networked Society," Ericsson AB, SE-164 80 Stockholm, Sweden, EAB-16:006659 Uen, Revision A, 2016.

[3] E. Borgia, "The internet of things vision: Key features, applications and open issues," vol. 54. Pisa, Italy: Elsevier B.V, Oct. 2014, pp. 1–31.

[4] BLE Nano-RedBear. Accessed 2016, August 10. [Online]. Available: http://redbearlab.com/blenano/

[5] N. Gupta, *Inside Bluetooth Low Energy.* Beaverton, Oregon, USA: Artech House, 2013, ISBN: 978-1608075799.

[6] OpenMote CC2538. Accessed 2016, August. [Online]. Available: http://www.openmote.com/shop/openmote-cc2538.html

[7] SparkFun Electronics. Accessed 2016, August 20. [Online]. Available: https://www.sparkfun.com/products/11697

[8] TCM5000 DATASHEET. Accessed 2016, July 25. [Online]. Available: http://www.epssilon.cl/files/EPS5000.pdf

[9] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *Proceedings of the 4th international symposium on information processing in sensor networks*, Los Angeles, California, USA, Apr. 15, 2005, pp. 364–369.

[10] *IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*, IEEE Std. 802.15.4-2011, 2011.

[11] 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. Accessed 2016, August 15. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2420.pdf

[12] Contiki: The Open Source Operating System for the Internet of Things. Accessed 2016, September 14. [Online]. Available: http://www.contiki-os.org/index.html#why

[13] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks.* Tampa, Florida, USA: IEEE, Nov. 16–18, 2004, pp. 455 – 462, ISBN: 0-7695-2260-2.

[14] *Standard Test Access Port and Boundary-Scan Architecture*, IEEE Std. 1149.1, 2013.

[15] Open On-Chip Debugger. Accessed 2016, September 18. [Online]. Available: http://openocd.org

[16] A. S. Sirotkin, "Debugging the linux kernel with jtag," vol. 23, no. 7, pp. 1–31, Dec. 2010.

[17] A. Robbins, *GDB Pocket Reference.* Sebastopol,California, USA: O'Reilly Media Inc, 2005, ISBN: 9780596100278.

[18] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source Level Debugger*, 9th ed. Boston, Massachusetts, USA: Free Software Foundation, 1999, ISBN: 1-882114-77-9.

[19] Arch Linux-arm-none-eabi-gdb. Accessed 2016, September 18. [Online]. Available: https://www.archlinux.org/packages/community/x86_64/arm-none-eabi-gdb/

[20] M. Doddavenkatappa, M. C. Chan, and A. Ananda, "Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed," in *Proceedings of the 7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2011).* Shanghai, China: Springer Berlin Heidelberg, Apr. 17-19, 2011, pp. 302–316, ISBN: 978-3-642-29272-9.

[21] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel, "Flocklab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems," in *Proceedings of the 12th ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN).* Philadelphia, USA: ACM, Apr. 8–11, 2013, pp. 153–166, ISBN: 978-1-4503-1959-1.

[22] V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "TWIST: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks," in *Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality.* Florence, Italy: ACM New York, May 26, 2006, pp. 63–70, ISBN: 1-59593-360-3.

[23] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A large scale open experimental IoT testbed," in *Proceedings of the 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. Milan, Italy: IEEE, Dec. 14–16, 2015, pp. 459 – 464, ISBN: 978-1-5090-0366-2.

[24] M. Nati, A. Gluhak, H. Abangar, and W. Headley, "Smartcampus: A user-centric testbed for Internet of Things experimentation," in *Proceedings of the 2013 16th International Symposium on Wireless Personal Multimedia Communications (WPMC)*. New Jersey, USA: IEEE, Jun. 24–27, 2013, pp. 1–6, ISSN: 1882-5621.

[25] Tinynode 184. Accessed 2016, September 15. [Online]. Available: http://www.tinynode.com/?q=product/tinynode184/tn-184-868

[26] R. Jurdak, K. Klues, B. Kusy, C. Richter, K. Langendoen, and M. Brunig, "Opal: A multi-radio platform for high throughput wireless sensor networks," *IEEE Embedded Systems Letters*, vol. 3, no. 4, pp. 121 – 124, 2011.

[27] IRIS Wireless Measurement System. Accessed 2016, October 20. [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf

[28] RIOT-The friendly Operating System for the Internet of Things. Accessed 2016, October 21. [Online]. Available: https://www.riot-os.org/#home

[29] FreeRTOS- Market leading RTOS (Real Time Operating System). Accessed 2016, September 18. [Online]. Available: http://www.freertos.org

[30] TinyOS Home Page. Accessed 2016, July 25. [Online]. Available: http://tinyos.net

[31] R. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, Californnia, USA, 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[32] WiringPi. Accessed 2016, July 20. [Online]. Available: http://wiringpi.com/

[33] SSH/OpenSSH/PortForwarding. Accessed 2016, July 20. [Online]. Available: https://help.ubuntu.com/community/SSH/OpenSSH/PortForwardin

[34] Bootloader (BSL) for MSP low-power microcontrollers. Accessed 2016, July 25. [Online]. Available: http://www.ti.com/tool/MSPBSL/

[35] SEGGER- The Embedded Experts. Accessed 2016, July 25. [Online]. Available: https://www.segger.com/jlink-debug-probes.html