ARM IoT Application Project
Final Report

Final Version

Name	Student number
Nemanja Vukota	2190646
Miika Keisu	2125909
Teemu Partanen	2190507
Matti Pulkkinen	2114842
Joonas Nikula	2190468

1. Executive abstract

In this project we explore the possibility to bring the Internet of Things into the ordinary people's homes and to make the lives of people a bit more digital and perhaps even easier. We targeted a simple MoccaMaster coffee machine as our apparatus which we would bring online. We used a few different sensors to measure the status of the coffee machine. Alongside with this, we utilized ARM's Device Server and implemented our own API between the Device Server and the client application. As a final result, we got a system that presented the status of the coffee machine in almost real time. We see that this kind of a system does have some potential to be used, especially in working environments where workers are required to make their own coffee. Future will show will the Internet truly be integrated into every house application possible.

Contents

1. EXECUTIVE ABSTRACT 2

2. INTRODUCTION 4

3. CUSTOMER 4

4. ACRONYMS AND DEFINITIONS 4

5. REFERENCES 5

6. PROJECT DESCRIPTION 5

6.1 Deployment 5

6.2 ARM mbed IoT devices 6

6.3 Device Server 6

6.4 Web service and API 7

6.5 Monitoring UI 8

7. REALISED REQUIREMENTS 8

7.1 ARM mbed IoT devices 8

7.2 Web service and API 10

7.3 Monitoring UI 12

8. USED SOFTWARE DEVELOPMENT METHOD 15

9. TESTING / ACCEPTANCE OF THE SOFTWARE 16

9.1 Client acceptance testing 16

9.2 System testing 16

10. PROJECT TIMELINE 17

11. FEEDBACK TO THE COURSE ORGANIZERS 17

12. PICTURES 18

2. Introduction

In this document we will go through our project in which we developed an Internet of Things application for our client using their mbed platform and Device Server. We will firstly introduce to you our customer, which was ARM. Then we will further continue to explain and introduce our project. What were our realized requirements? Which software development method did we choose to use and why? We will also show how did we evaluate the code and the project overall. Finally, we will explain about our demo meeting we had with our customer, which worked as an acceptance test for the application as well.

3. Customer

ARM is the world's leading semiconductor intellectual property (IP) supplier. They design scalable, energy efficient-processors and related technologies to deliver the intelligence in applications ranging from sensors to servers, including smartphones, tablets, enterprise infrastructure and the Internet of Things. Since the company began in 1990, their partners have shipped over 60 billion System on Chips (SoC) that contain ARM IP. ARM's business model is based on designing and licensing of IP rather than manufacturing selling of actual semiconductor chips. [1]

Our client's office and contact persons were located in Oulu. ARM arrived in Oulu by acquiring Sensinode Oy in 2013. Sensinode was a company that has led the creation of the 6LoWPAN and CoAP standards for low cost low power devices, and has been a key contributor to the IETF, ZigBee IP, ETSI and OMA standardization efforts. ARM continues to develop IoT technologies to accelerate the Internet of Things and support 30 billion connected devices by 2020. [3]

4. Acronyms and Definitions

IoT	Internet of Things
IP	Intellectual Property
SoC	System on Chip
CoAP	Constrained Application Protocol
IR	Infrared
I2C	Inter-Integrated Circuit
DHCP	Dynamic Host Configuration Protocol
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
ETSI	European Telecommunications Standards Institute
OMA	Open Mobile Alliance
M2M	Machine-to-machine
OMALWM2M	OMA lightweight M2M
MQTT	MQ Telemetry Transport
DTLS	Datagram Transport Layer Security
TLS	Transport Layer Security

5. References

[1] http://www.arm.com/about/company-profile/index.php, quoted 6.6.2015.
[2]http://www.arm.com/about/newsroom/arm-acquires-sensinode-oy-to-accelerate-the-internet-of-things-and-support-30-billion-connected.php, quoted 21.6.2015.
[3] http://www.nxp.com/documents/leaflet/LPC1768.pdf
[4] https://developer.mbed.org/components/MLX90614-I2C-Infrared-Thermometer/
[5] http://www.arm.com/products/internet-of-things-solutions/mbed-IoT-device-platform.php

[6] https://www.python.org/dev/peps/pep-0249/

[7] Raccoon, L. B. S. "The chaos model and the chaos cycle." *ACM SIGSOFT Software Engineering Notes* 20.1 (1995): 55-66.

6. Project Description

Already after the ARM's introduction of the project it was clear that the project would be all about Internet of Things. The goal for the project was and still is to spread the word of IoT and make more and more developers to start developing their own IoT applications using ARM's devices. This was also the reason why ARM asked us to host the project on GitHub under Apache License. Everyone could go there and see what we did. We also did a short video about our system to promote it even more. ARM also required us to use their Device Server to host the resources of the mbed chips.

In the practice, the project started with one month long brainstorming period in which we were tasked to come up with 2-3 great IoT ideas. Then we presented the ideas to the ARM representatives and together with them we chose to implement the IoT Coffee Machine. Probably the biggest reason for this choice was that the idea was very fun and in this way it could attract even more attention from the community. Also it was very doable and we predicted that there wouldn't be any major issues alongside the road.

After we had chosen the idea to go with, we started to design and implement the system. All the hardware necessary for the project was provided to us by ARM including the original Moccamaster coffee machine.

6.1 Deployment

The system was deployed as a composition of 5 software and hardware parts. First we have the ARM IoT devices that implement the first and the second part of the system. The device in question is an mbed NXP LPC1768 [3] development board that runs on an ARM Cortex-M3 microcontroller. The system uses two of these devices, two application boards (development boards with various different sensors and devices attached), two WiFly WLAN adapters and some external electronics and sensors to implement a coffee pot monitoring device and an alarm device.

The third part is the mbed Device Server that handles the connections from IoT devices providing a uniform HTTP interface for external applications to access the data provided by mbed devices.

The fourth part of the system is a web service that implements an API that provides the user interface application with both live and history data and collects and process sensor data fetched from the Device Server.

The last part of the system is a web application that presents the state of the monitored coffee machine and history data based on data received from the API.

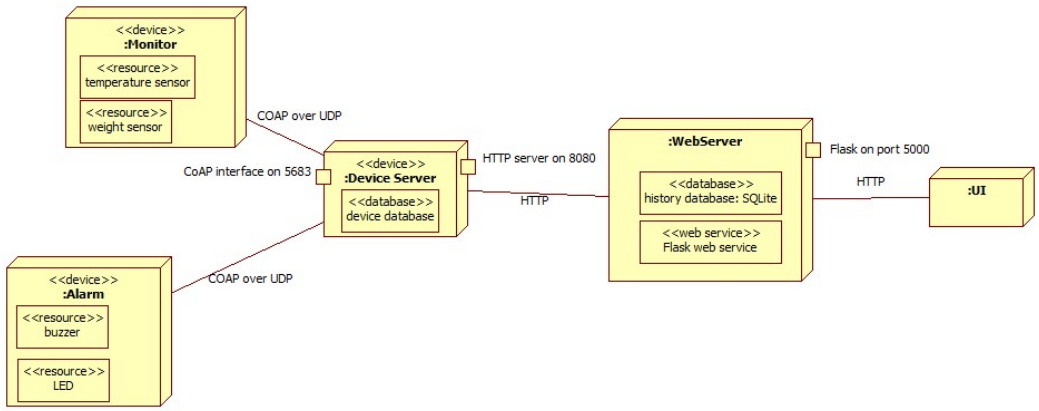


Figure 1: Deployment diagram

6.2 ARM mbed IoT devices

The devices we used in the system are NXP LPC1768 development boards that were equipped with application boards and WiFi adapters. These devices were used to implement a monitor platform and an alarm device.

The sensors and IO devices used in the system were implemented as asynchronous resources that the Device Server can display to applications and forward received HTTP requests as CoAP requests to the required resource. To implement the features that were required, we implemented 4 different resources: Temperature, weight, buzzer (alarm) and a LED.

The monitor device uses resistive pressure sensors and a voltage divider circuit to measure the voltage in a node using an analog input. The device is also connected to a digital IR temperature sensor that provides the temperature of the pot using I2C bus.

The alarm device can be controlled with the buzzer and LED resources to set the alarm and fetch the state of the buzzer and LED. When the alarm is set buzzer starts ‘beeping’ and bright red LED is turned on. At the same time the LCD screen of the application board displays an out of coffee message.

6.3 Device server

Device Server handles the connections from IoT devices. The Device Server's role is to provide an uniform HTTP interface for external applications to access the data provided by mbed devices. It utilizes open source protocols like CoAP/HTTP, MQTT, TLS/TCP, DTLS/UDP and OMALWM2M for data communication and device management. [5] Device Server provides following features and benefits:

- Support for key IoT standards enables interoperability: OMA Lightweight M2M, CoAP, HTTP, MQTT, DTLS, TLS
- Built-in security management, load balancing and distributed clustering
- Resource discovery and group support
- Caching and subscription aggregation support
- C, Java SE and Java ME Clients for IoT devices

- Reduced time-to-market
- Provides strong end-to-end trust and security for constrained devices and networks
- Enable device management and application data with the same solution
- Typical 10x reduction in bandwidth
- Enables access to the ecosystem of ARM IoT devices

Front page of the ARM mbed Device Server is displayed in the figure 2.

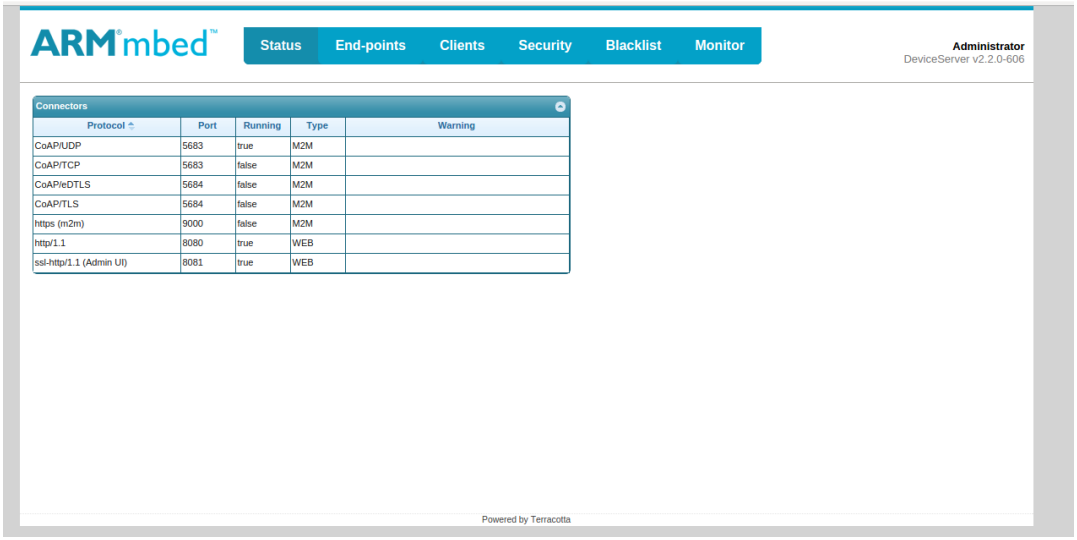


Figure 2: Front page of the Device Server

6.4 Web service and API

Since we wanted to store historical data and also manipulate the raw sensor data coming from the Device Server, it was an obvious choice to implement a web service as a part of the system. For hosting the web services and the database, we asked a virtual server machine from the course staff.

The web service and the API is implemented with Python. Furthermore the API uses Flask micro web application framework. Flask was an easy choice for this project, since it's under BSD license and allows easy RESTful API creation with Python. Our team also had previous experience from the Flask framework.

All the permanent data is stored in SQLite database. SQLite is a small RDBMS that is directly linked to the application and no external database connection is needed. In SQLite, the entire data can be stored in a single file on the hosting machine.

The key component of the web service is the *server program* that is responsible for fetching and manipulating the sensor data from the Device Server and storing it into the database. Server is also responsible for switching the Coffee Monitor alarm on or off whenever needed.

The web service communicates with the Device Server RESTful API via HTTP requests. The algorithm fetches sensor data from the *weight sensor* and the *temperature sensor* every three seconds and stores the data in the SQLite database.

Based on that raw sensor data, the algorithm then calculates the amount and temperature for the coffee and also gives a current status (i.e. “pot is full”) for the machine and stores all that information in the database as well. If the coffee pot gets empty, server sends a PUT request to the alarm endpoint in the Device Server and sets the alarm on to inform that the system is out of coffee. See Table X.

The API is responsible to serve our HTML web client (UI) application. The API is RESTlike, and has two resources: *main (/)* and *history (/history).* These resources offer current and historical data about the coffee machine state (from the database) for the web client to consume. Data is provided in JSON format.

6.5 Monitoring UI

Monitoring UI is a client, which is used to present the current status to user. The monitor fetches status information from API. The monitoring UI is implemented using HTML, CSS and JavaScript. Both desktop and mobile versions of the monitor are implemented.

The monitor provides the information about the coffee amount in pot, pot’s temperature and status. The monitoring system doesn’t do any data processing; it merely presents the values got from the API. The amount is presented as a percentage in five levels; 0 %, 25 %, 50 %, 75 % and 100%. The temperature is presented as an integer in Celsius degrees. The status is verbal information of the pot’s current status. Currently the options are; ‘no coffee’, ‘cold coffee’, ‘¼ pot left’, ‘½ pot left’, ‘¾ pot left’ and ‘full pot’. The previous values amounts and temperatures are also presented in a line chart.

7. Realised requirements

In this part of the document we describe what did we actually implement and how, in detail. We will go through each of the component required by the system. This includes the hardware and Device Server, the API and the UI of the system.

7.1 ARM mbed IoT devices

The monitoring device was implemented with 2 separate resources, temperature and weight. The realization of the sensors that were paired to these resources required us to implement some external electronics. For the weight sensors, the circuit has been equipped with a small capacitor to filter out some noise. In addition to this, we needed to implement a voltage divider circuit so we could measure the voltage at the node after the sensor. The temperature sensor was implemented using a digital IR temperature sensor attached to the LPC1768 I2C bus [4]. For the temperature sensor we needed some pull-up resistors for the I2C bus. In addition to these we soldered together a lot of wires to enable us to position the sensors to the coffee maker. The circuit required to implement these sensors is displayed in figure 3.

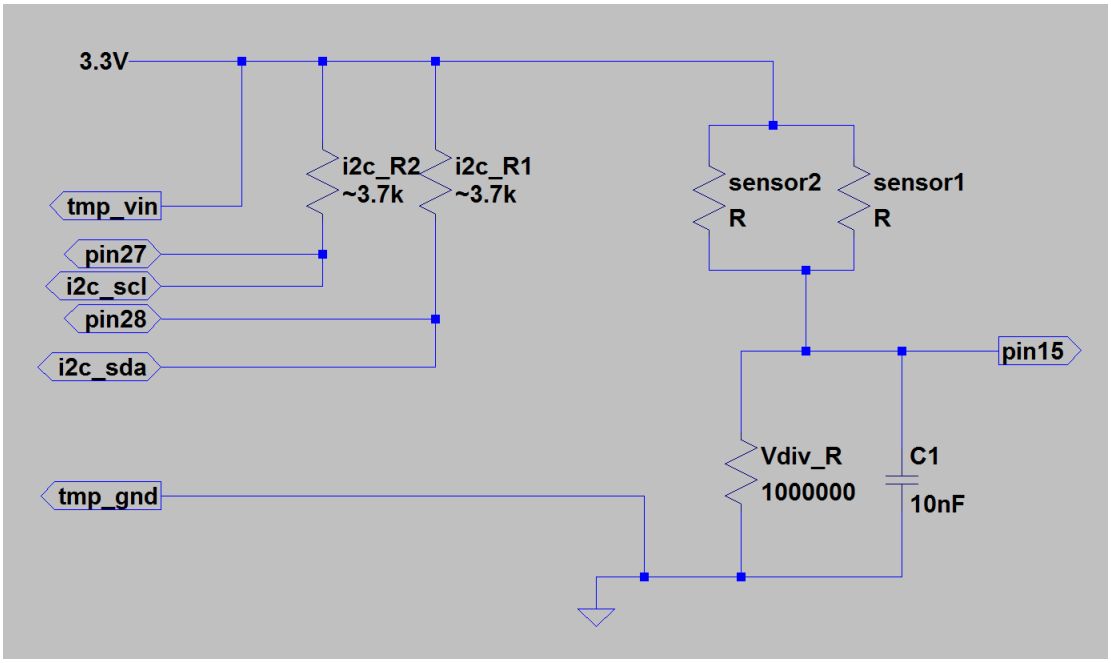


Figure 3: Circuit diagram

On the software side the weight resource calculates a mean value over 500 samples taken from the sensor when a call to the resource is made. We perform no scaling or conversions for the measured voltage on the device because we decided to perform these operations on the server side to keep the device code as simple as possible.

The alarm device was also implemented with two resources, which are used to power on the LED and the buzzer on the application board via HTTP PUT. Buzzer and LED statuses can be read with a HTTP GET. When the Buzzer is set on it will beep for 5 seconds at 700Hz frequency.

There are a few steps that need to be taken to setup the system on the devices. First you need to connect the wifi-module, application board and the electronics for the monitor device. The second step is to go to the device source code and change the values of a few parameters to match the desired setup. These parameters are monitor, alarm, server address, server port and DHCP or ip configuration. If DHCP is enabled and network supports DHCP, the ip configuration is not necessary. Monitor and alarm-parameters change the operating mode of the device to monitor, alarm or multifunctional device. The server address and port need to be set to match those of a valid Device Server. After this you need to compile the source code and upload it into the device. During power on, devices will attach to the network and register themselves into the Device Server.

Because we were unable to get server that would suit our needs in time we decided to install the ARM Device Server software on Raspberry Pi, which was located at one of our team member's home. Device Server on the Raspberry Pi was accessible over the Internet. When devices are registered to the Device Server as end-points resources from the devices can be accessed via the Device Server with HTTP requests. Resources and their URIs implemented are described in the table 1. Example of how the RESTful API can be used with complete system is illustrated in a sequence diagram in figure 4.

URI	METHOD	DESCRIPTION
/alarm_led/	GET	Get LED status (on or off)
/alarm_led/	PUT	Set LED on or off
/alarm_buzz/	GET	Get buzzer status (on or off)
/alarm_buzz/	PUT	Set buzzer on or off.
/sen/weight/	GET	Get weight
/sen/temp/	GET	Get temperature.

Table 1: REST API resources provided by the mbed devices.

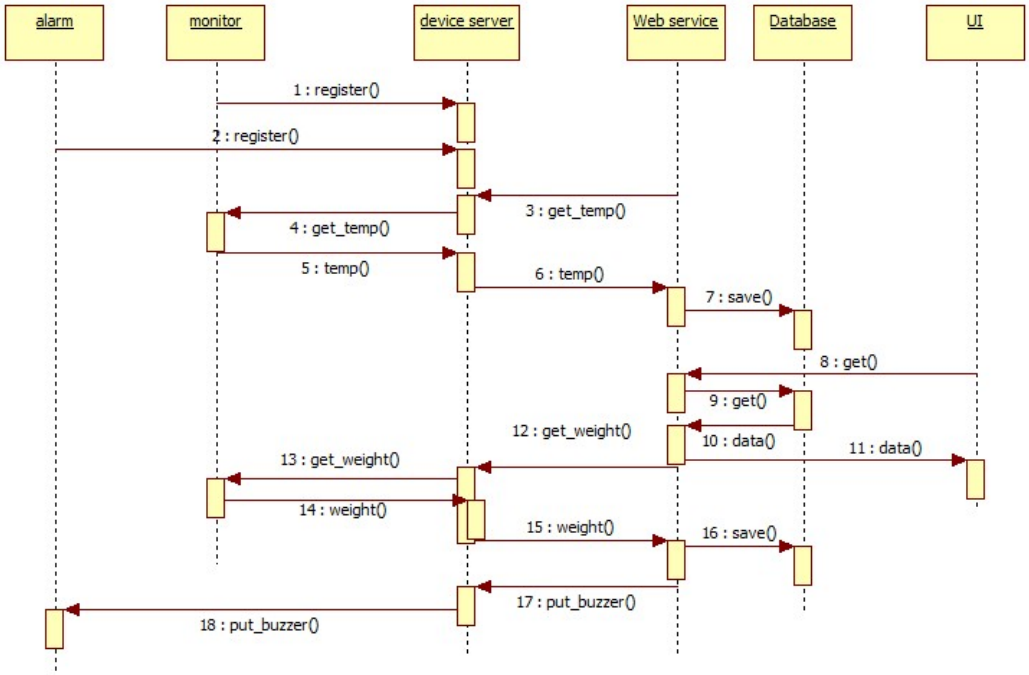


Figure 4: Sequence diagram

7.2 Web service and API

The web service has three key components:

1. SQLite database (swp.db)

One database file to store all persistent data. Table description can be found in tables 2 & 3.

COLUMN	DATATYPE	DESCRIPTION
temp	REAL	Current temperature sensor value (raw data)
weight	INTEGER	Current weight sensor value (raw data)
timestamp	TEXT	UNIX timestamp

Table 2: Stores raw sensor data values fetched from the device server in every three seconds

COLUMN	DATATYPE	DESCRIPTION
temp	REAL	Temperature value (in celsius)
amount	INTEGER	Amount of coffee (0-100%)
status	TEXT	Textual description of coffee the status
timestamp	TEXT	UNIX timestamp

Table 3: Stores values calculated by the algorithm and furthermore API delivers the values to the client

2. Server program (server.py)

Server program, implemented with Python, is responsible for fetching current sensor values from the Device Server for every three seconds by using *Sensor* class. Fetched sensor data is stored in the *data* table with proper UNIX timestamp. Server program also runs an algorithm, that is responsible to create new rows in *api* table. These rows are generated based on the sensor data from *data* table. The algorithm converts the raw *weight sensor* value in four-step (0, 25, 50, 75 or 100%) scale to show the approximate amount of the coffee in the pot. Limit values are defined as GLOBAL variables in the beginning of the file, so they are easily modified if needed.

Temperature sensor value is used as it. With the combination of amount and temperature values the status message is also decided. All these three values with the current UNIX timestamp are then stored in the database *api* table.

Server program has also function that toggles the alarm led and buzzer if the amount of coffee gets zero. Alarm led and buzzer is operated via Device Server resources by using GET method to get the status and PUT to set the value of the led/buzzer (1=on, 0=off).

3. RESTful API (api.py)

The API is implemented in Python using Flask web framework. API serves the client (UI) application. API has two resources: main (/) and history (/history) that are described in table 4. Main returns the newest row from the *api* table and history the 100 latest rows.

GET, 200 OK

```
{
"amount": 50,
"status": "1/2 pot left",
"temperature": 80,
"timestamp": 1433677554
}
```

Python programs use Python's sqlite3 SQLite module to connect to the database. The module provides a SQL interface compliant with the Python DB-API 2.0 specification [6].

RESOURCE	METHOD	DESCRIPTION
/	GET	Shows the newest from the API table.
/history	GET	Shows 100 newest rows from API table.

Table 4: API Resources

7.3 Monitoring UI

The monitoring system is web based and implemented using HTML, CSS and JavaScript with jQuery library. For plotting purposes Google Visualization API used and a CoffeePot library was implemented for this project. CoffeePot library is used to draw a MoccaMaster like coffee pot to a canvas and then filled by percentage. It was designed by drawing the pot to a graph paper from a picture. After that the relative coordinates were calculated from the origin. The graph paper is presented in figure 5.

When the page is loaded the system requests history data from the API and fills the line chart with it. After that the monitoring system requests new values from the API every five seconds and redraws the plot after five requests. The plotting interval was implemented to make it easier to read the line chart. The line chart holds one hundred amount and temperature values. All these presented settings are parameterized and can be changed easily from the logic.js.

The system has two different views; normal view and alternative view for mobile devices. Normal view is presented in figure 6 and alternative view is presented in figure 7. Both views contain the same information but the elements are arranged differently. The view is automatically switched to alternative view if the browser width is less than height.

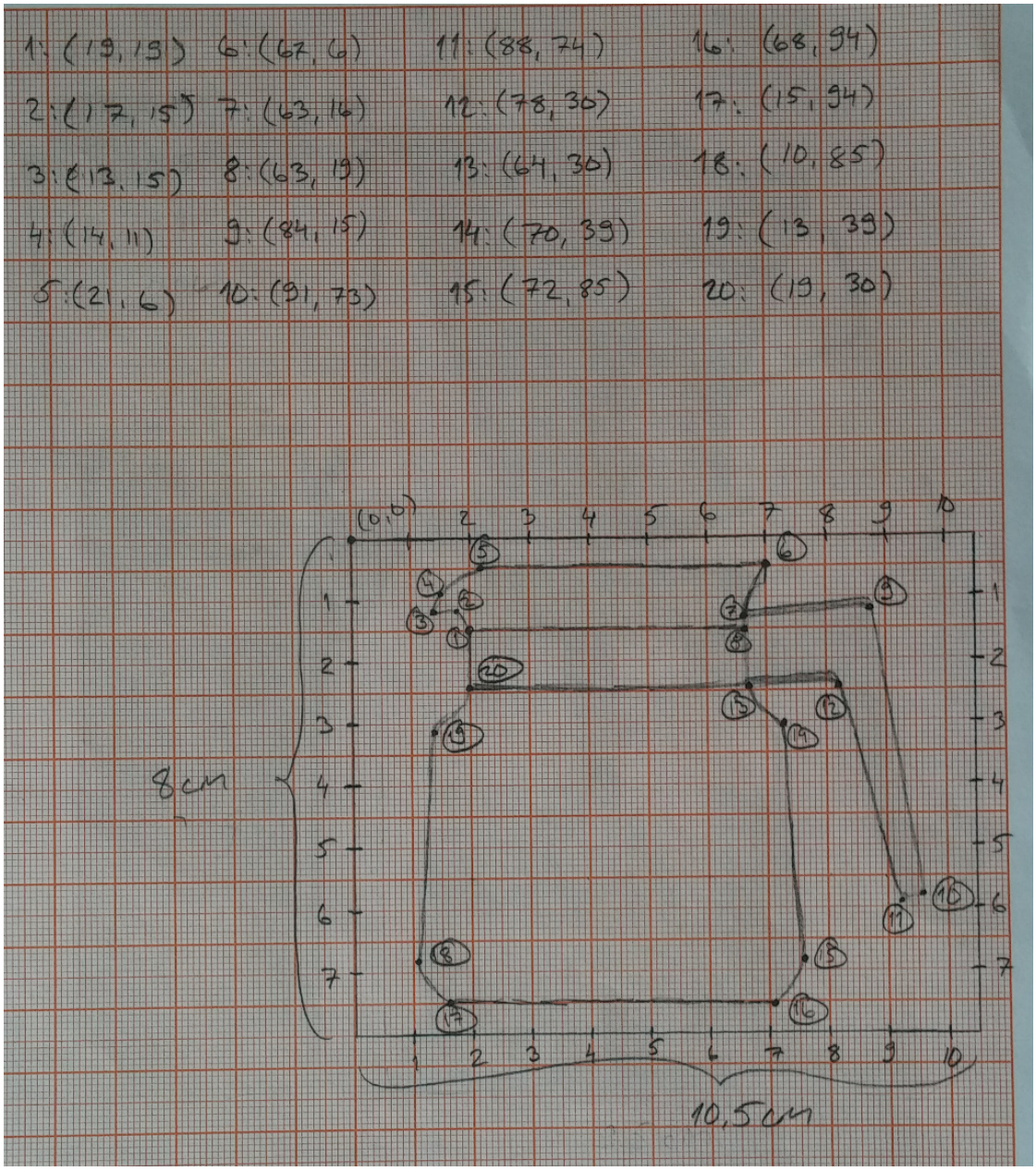


Figure 5: Original design of the JavaScript coffee pot.

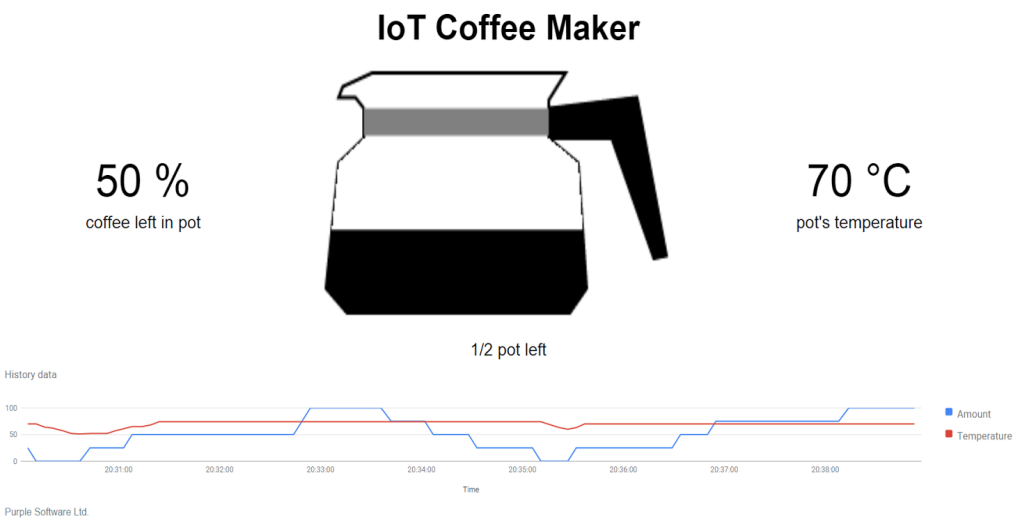


Figure 6: Monitoring system's normal view.

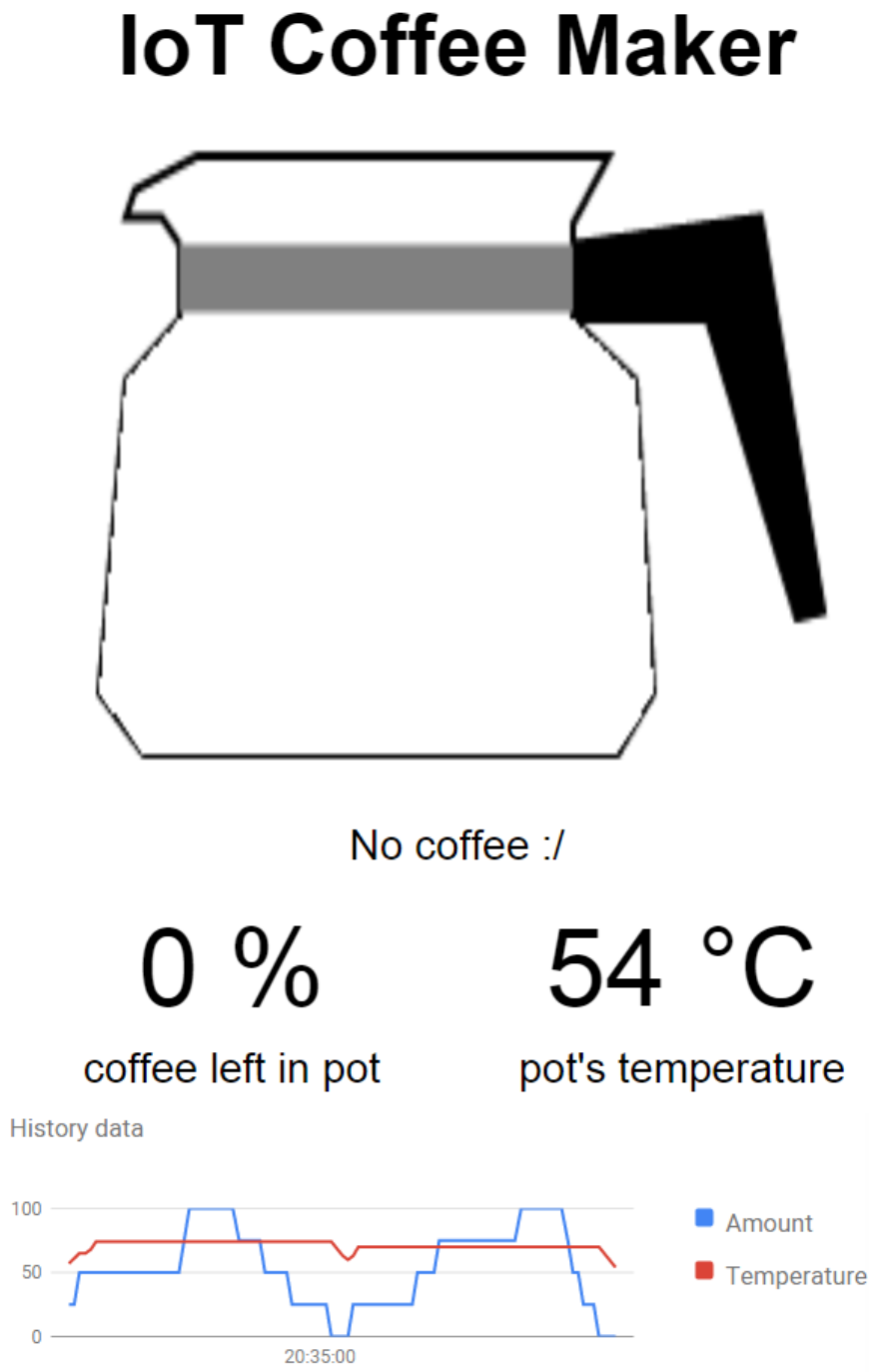


Figure 7: Monitoring system's alternative view.

8. Used software development method

Quite soon after the beginning of the project we found out that our individual schedules didn't go quite well together. Because of this we had to start following more or less the Chaos Model as our software development method. Quite frankly, it was the only one doable by us. Basically Chaos Model [7] attempts to bridge the gap between traditional project management models such as spiral model and waterfall model and programming methodologies like agile programming. As we had quite tight schedule but also had to fix and solve technical issues rather quickly, we ended up following the Chaos Strategy of the Chaos model. Main rule in Chaos Strategy is

that the most important issue needs to be solved first. What this meant for us in practice is that before we started implementing stuff, we divided specific areas of the system to each one of us. After that we started to implement the individual tasks we got and to solve issues that came in front of us as soon as possible. At the end we integrated each part of the system into the whole system. After the integration of the system we needed to calibrate and test the system.

9. Testing / acceptance of the software

Quality of the software was assured by making so-called sanity check during implementation phase i.e. manually checking that the system works before submitting changes to the code repositories. We also performed client acceptance testing and small unit tests to assure the quality of the software.

9.1 Client acceptance testing

We conducted acceptance testing with our clients at the ARM office in Oulu to test if the requirements of our specification and given task were met. We showed the ARM staff our system working as a whole in real-time, in a real environment.

We demonstrated the functionality of our system by making a full pot of coffee while showing the monitoring and web application UI in the action. During the test all connection was handled through the Internet to show that the system can also cope with network latency and other challenges that might occur. In the test following steps were taken:

- Make full pot of coffee and confirm that the UI shows 100% and full pot
- Pour off some of the coffee and confirm that the UI shows real amount of coffee left in the pot
- Take off the pot, confirm that alarm is set.
- Put the pot back and confirm again that right amount of coffee is displayed in the UI.

Our clients at ARM were very pleased with the demo we arranged and how the system worked, as it seemed to be just as accurate as we hoped. Our clients also stated that we took quite a risk when routing all the connections through the Internet but still managed to provide a reliable live demo.

9.2 System testing

Since our system included the usage of two different sensors, the thermometer and the pressure sensor, we had to do a lot of the calibration for them. Alongside with this it was convenient to do the manual integration tests for the system.

Basically with the calibration we had to start from the bottom. We boiled countless amount of water through the coffee machine in order to iterate the positions and sensitivity of the sensors. Especially the pressure sensor proved out to be quite difficult to calibrate since it wasn't sensitive enough for our coffee machine. But after many boiled waters we managed to calibrate the sensors so that they showed quite nice values.

Alongside the calibration of the sensors, we also made sure that the API and the UI of the system worked as supposed. Especially the API, which worked between the Device Server and the UI, needed tweaking a lot. We also added few things to the API after we observed that it was needed. For example we added the average valuation to the API to make the pressure sensor values a bit more rational and sensible.

Furthermore, we implemented simple unit tests for the resources that were hosted on

the hardware. We checked the HTTP response codes and the connection overall.

10. Project timeline

Project started on the 27th of January with a kick-off meeting with ARM at their office. ARM introduced the project and requirements, which they wanted to be met. In this meeting we also agreed to start of with a brainstorming period of about one month. In this time we should come up with 2-3 great and innovative IoT ideas.

Our idea, the IoT Coffee Machine, was accepted about a month later and the actual design and the implementation of the system started at the end of February/beginning of March.

The testing phase of the system began in the April and ran to the middle of May.

19th of May we held a client acceptance meeting with ARM where we introduced and demoed our system.

During the end of May and June we concentrated on writing this final report. Also, we installed our system on a Raspberry Pi that was given to us by ARM. They want to start using the system at their office.

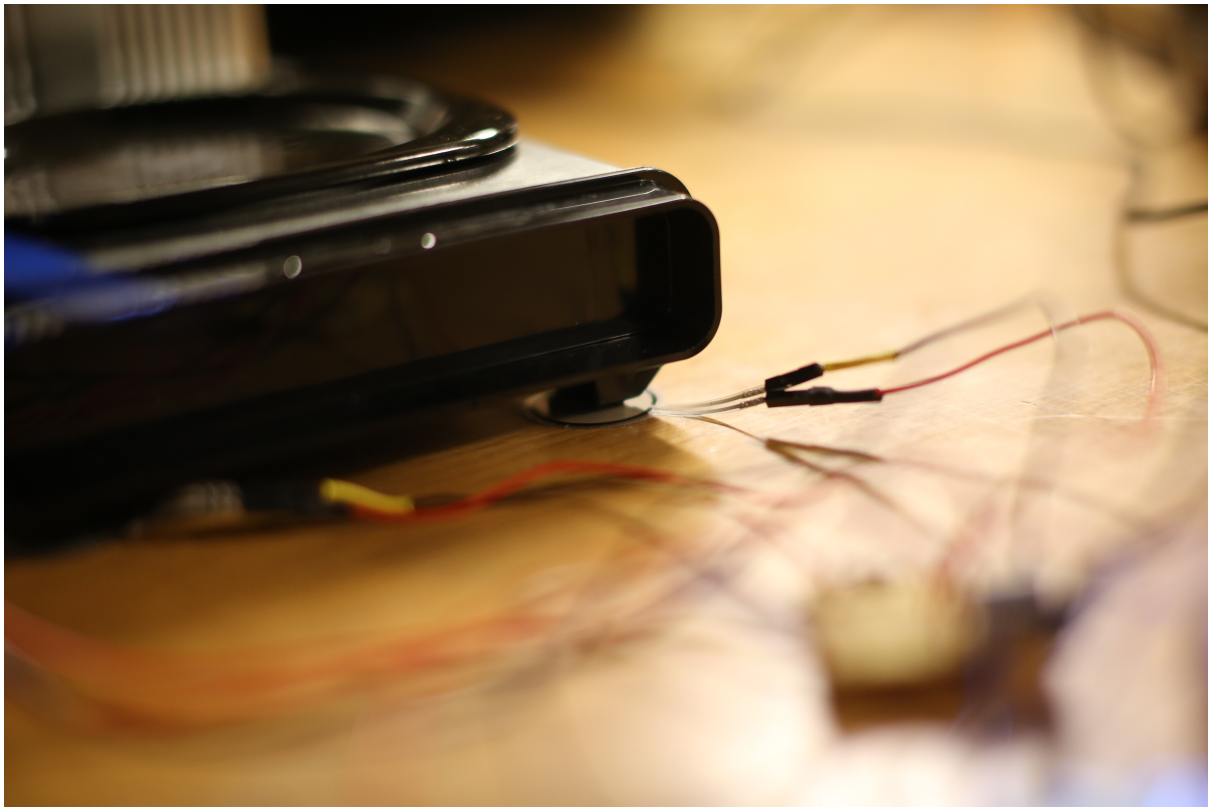
Finally, on the 7th of July we wrapped the project.

11. Feedback to the course organizers

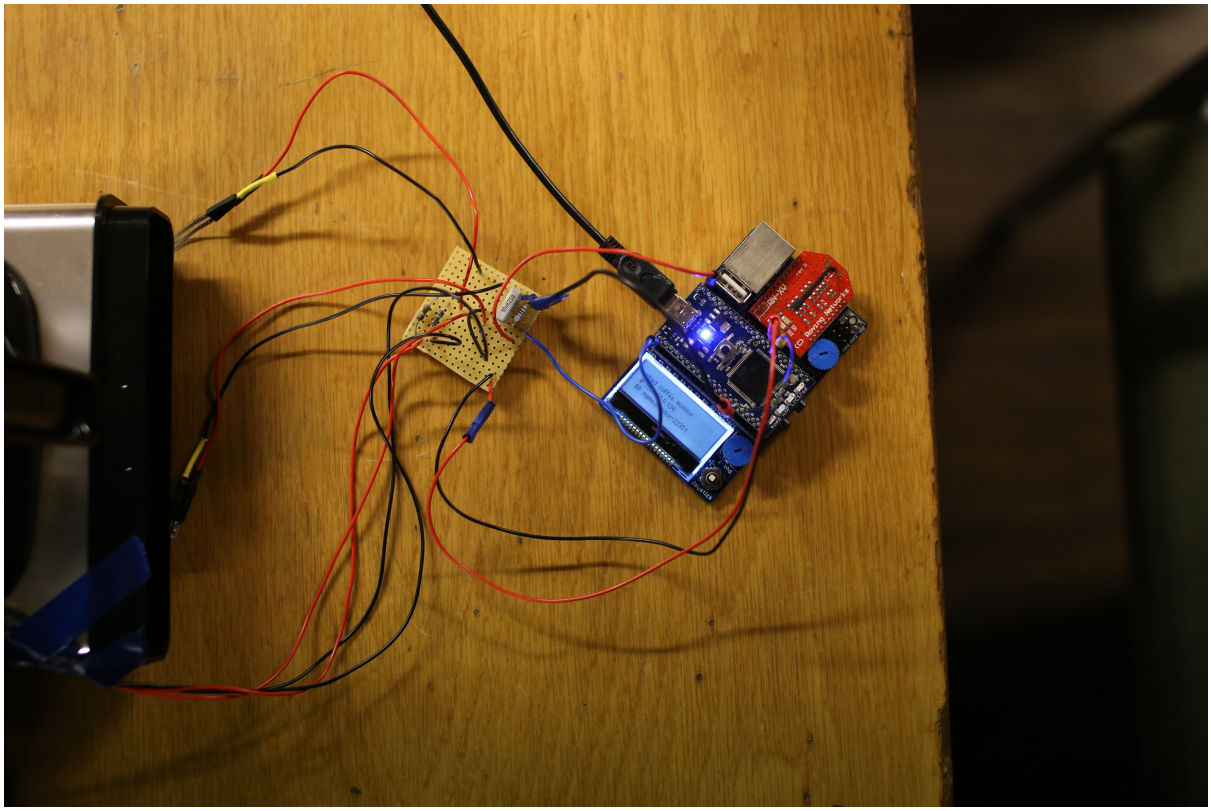
We liked the course and generally like these kinds of project courses. This one was especially good one because of the freedom that the students had. Also, working with a real company and not only for university was a big plus. We have been worked as a group before also, so it was a real joy working with the same group again. These kinds of courses develop the students in the best way for working life. Great course and thank you!

12. Pictures

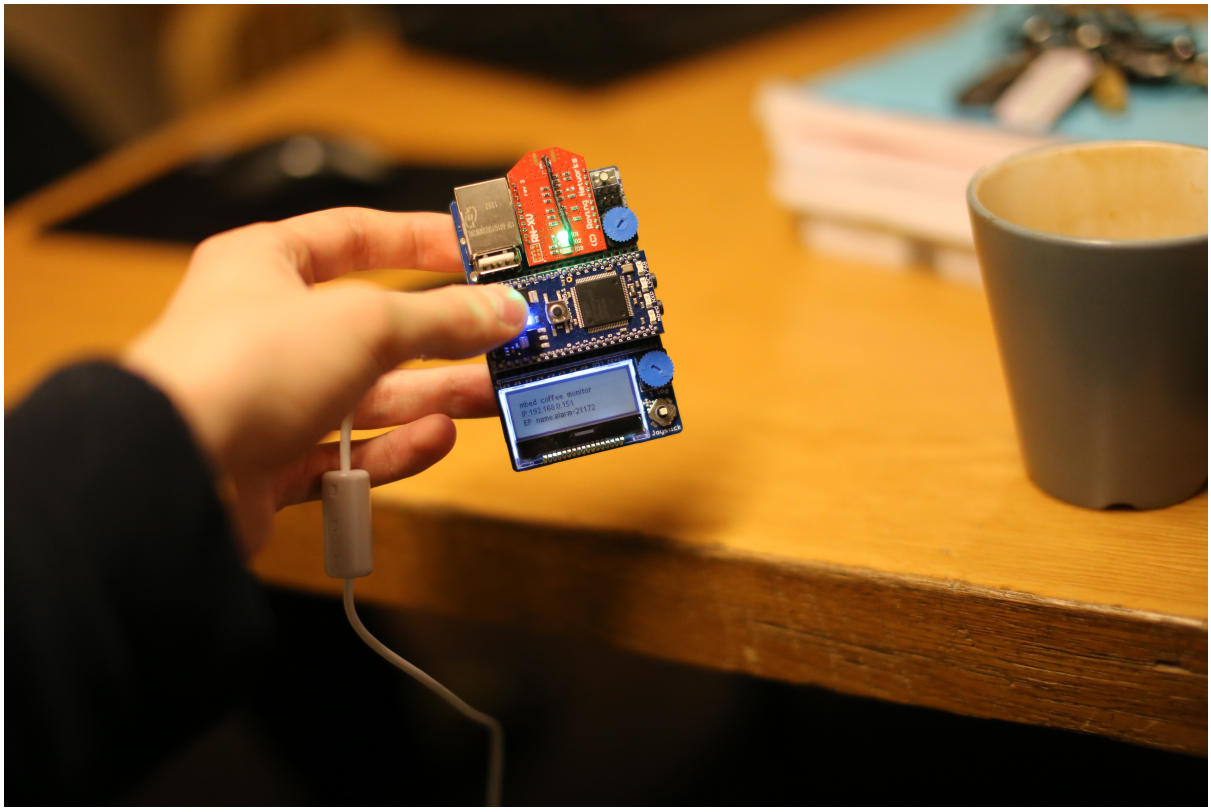
Picture 1: Heat sensor



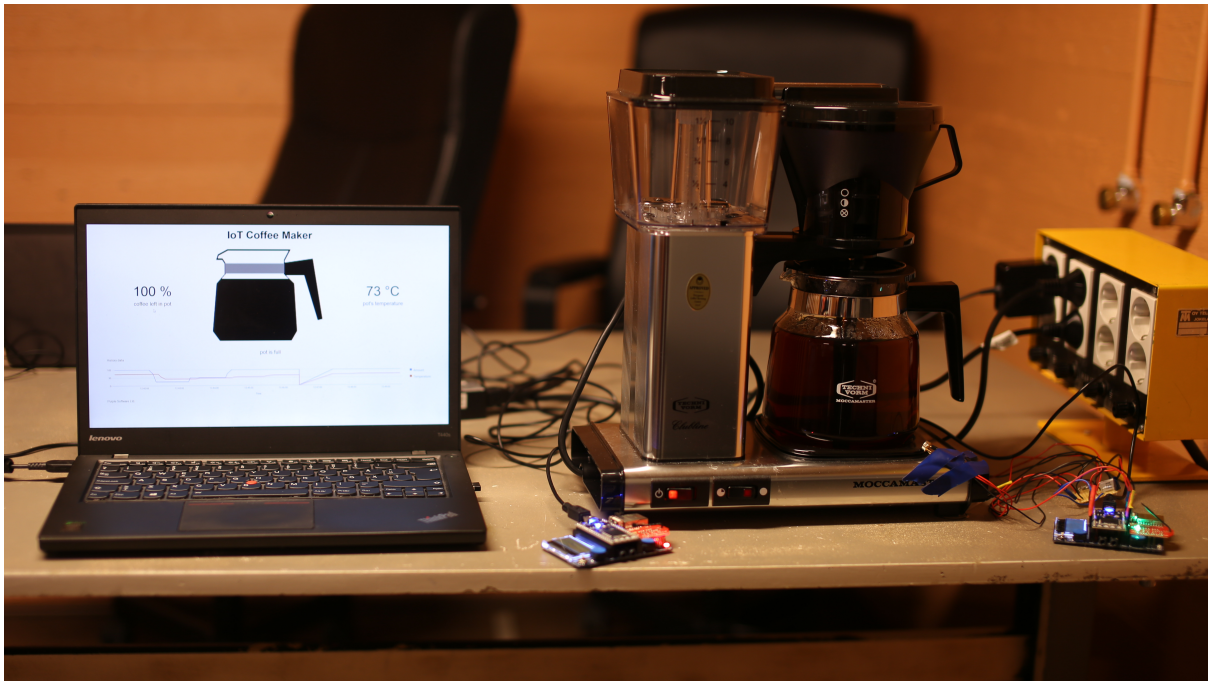
Picture 2: Pressure sensors



Picture 3: Mbed chip + application board



Picture 4: Mbed chip + application board



Picture 5: System in use