Processing of temperature using WEMOS D1 mini

Aljaž Blažej and Žan Ožbot

Wireless Sensor Network course research seminar report

May 12, 2019

Mentor: prof. dr. Nikolaj Zimic

The world of IoT (Internet of Things) devices has seen a massive expansion in the past decade and found the use in countless fields like medicine, smart homes, cars, smart cities, wearable technology and many more. The crucial part with these devices is the inter-connectivity, which enables them to send the collected data to the servers, where it can be further processed. The article covers different ways of communication and interaction between devices and demonstrates a real world use case. It also describes the problems one can expect to come across while developing in this field and the workarounds one can take to still achieve the initial goal. The last part of the work is focused on the results we gathered while measuring the temperature with the micro-controller and a graphical representation of the acquired data.

Wireless sensor network | MQTT protocol | AWS IoT | HTTP server | nRF24 module | reading of temperature

This article demonstrates how to set up your own ESP8266 WiFi-enabled micro-controller (in our case WEMOS D1 mini) to read the temperature, send it to the MQTT broker and run the HTTP server alongside, which shows the real time temperature without the need to subscribe to the MQTT topic. We achieved that by using the FreeRTOS operating system.

The project was started by using two micro-controllers. One of them acted as a master, which ran a HTTP server and sent data to the MQTT broker, and the other one as a slave. The later collected the temperature data and sent it over the wireless transceiver to the master using the nRF24 module. The problem in this scenario was that the programming languages C and C++ have some compliance issues and we could not merge the HTTP server and the nRF24 code base. We decided to drop the idea of using multiple micro-controllers and focused on creating a solution for one only.

Our current micro-controller operates all the following task (i) collects temperature data using the BMP280 module, (ii) runs a HTTP server with WebSockets to show real time temperature changes, and (iii) sends the collected data to a topic using the MQTT protocol. Along the way of setting all that up, we had a problem with securing the sent data. We tried securing it with certificates but as it turns out the processing of them takes way too much memory, which causes the task to hang. We gave up on that option as well and went with using plain old username and password strategy for accessing the MQTT broker.

The article contains (i) all the methods we used while building our solution even the ones that were abandoned for any of the previously mentioned reasons, (ii) the problems one may encounter and possible fixes, (iii) different measurements while playing around with the temperature sensor, and (iv) discussion about the future work.

1. Methods

All the methods used in the process of creating the final solution are described below.

Data collection. The first thing we did is to decide which sensor to use. Since we had a lot of ideas that included the use of the temperature, we decided to use the BMP280 module. We tested a lot of them and they can be found in the 3. section (i.e. results).

The sensor is connected to the micro-controller using the I2C, so it is important for us to enable it, otherwise the sensor may not work properly. We do that using the following two lines of code.

```
i2c_init(BUS_I2C, SCL, SDA, I2C_FREQ_100K);
gpio_enable(SCL, GPIO_OUTPUT);
```

Then we make sure we add the I2C imports to the Makefile.

```
EXTRA_COMPONENTS=extras/i2c
```

After that, we configure the BMP280 using the provided sample and we are ready to set up the task which reads the temperature repeatedly, in our case every 2 seconds. The temperature data is stored in a global variable so it can be accessed in the HTTP server as well as in the MQTT script.

Procesiranje temperature z uporabo ploščice WEMOS D1 mini

Svet interneta stvari (angl. Internet of Things) je v zadnjem desetletju doživel velik razpon. Te pametne naprave lahko danes najdemo na številnih področjih, kot na primer v medicini, pametnih domovih, avtomobilih, pametnih mestih in drugje. Ključen del naprav je njihova povezljivost, ki jim omogoča pošiljanje zbranih podatkov na spletne strežnike, kjer so nato dokončno obdelani. Članek opisuje različne načine komunikacije in interakcije med napravami in praktičen primer te tehnologije. Prav tako opisuje probleme, s katerimi se lahko srečamo tekom razvoja in rešitve, ki nas kljub temu pripeljejo do zadanega cilja. Zadnji del članka je posvečen rezultatom meritev in grafičnemu prikazu temperature, ki smo jih izmerili z mikrokrmilnikom.

Brezžična senzorska omrežja | protokol MQTT | AWS IoT | strežnik HTTP | modul nRF24 | branje temperature

```c
// bmp.c
float temperature = 0;

// http_server.c, mqtt_client.c
extern float temperature;
```

Sending data with the nRF24 module. In the beginning we decided to use two micro-controllers so we needed a way to transfer the temperature data. We also wanted a wireless communication, so we could put the micro-controllers further apart and perform experiments in different closed environments. We decided to use the nRF24L01 [1] wireless transceiver module, which was already soldered to our micro-controller. It is designed for low power short distance data transmission. It uses the ISM frequency band at 2.4GHz and a proprietary communication protocol. We used the RF24 library and borrowed a sample implementation of the transmitter and receiver. We changed it so it transmitted temperature data on 1 second intervals. Although the implementation worked, we abandoned it for the reasons described later in the report.

HTTP server. To better carry out the experiments, we wanted to have a real time visual representation of the measurements. We decided to put a HTTP server on the master micro-controller to serve measurements to devices on the same network. We also wrote an HTML page with a graph, that uses WebSockets to show real time information without the need to refresh the page.

WebSocket. WebSocket is a communication protocol which uses TCP connection to send real time data from the web server to the web browser with low overhead. The connection can stay open for longer periods of time which makes the protocol useful for applications like ours.

Page details. The main part of the page was occupied by a graph that shows the changing of temperature over time. We set the scale of the temperature from 0°C to 100°C as the temperatures outside of the spectrum may potentially damage the controller. The graph was set to show 10 minutes of temperature data sequence, which allowed us to see temperature changes and patterns. Examples of the web page and the graph can be seen in the results section 3.

Sending and receiving data using MQTT. Firstly, we present in short the MQTT protocol and why it is used. Secondly, we demonstrate how to simply set up an AWS IoT account and how you can easily connect a sensor, which is in AWS world called a Thing. Lastly, we talk about why we switched from AWS IoT to another provider.

MQTT protocol. MQTT is a connectivity protocol, which was designed to be extremely lightweight publish/subscribe messaging transport [2]. Our micro-controller is using this protocol to publish the temperature data to a topic which is connected to the chosen MQTT broker service. Other clients can then subscribe to this topic and use the temperature information for their needs.

AWS IoT. Amazon Web Services (AWS) is an on-demand cloud computing platform offering multiple services, one of which is the IoT with the support for MQTT. To set up the message broker you can use either the command line interface or the provided UI. The following setup process uses both.

The process starts by creating a Thing, which can be done in the AWS IoT Mangage section. After entering the name, you can leave the other settings set to their default values. Secondly, gett get the Thing's endpoint address in the *Interact* tab. The process is followed by creating a private key and an ECC-based client certificate [3], which enables the platform to authenticate the device. To generate the private key and the certificates, run the following shell commands.

```
$ openssl ecparam -out ecckey.key -name prime256v1 -genkey
$ openssl req -new -sha256 -key ecckey.key -nodes -out eccCsr.csr
$ aws iot create-certificate-from-csr --certificate-signing-request
    file://eccCsr.csr --certificate-pem-outfile eccCert.crt --set-as-active
```

You can then convert the certificate to a C string to use it in the configuration files:

```
$ cat ecckey.key | sed -e 's/^/"/g' | sed -e 's/$/\\r\\n"/g'
```

The next step is creating an access policy and attaching it to the certificate.

```
$ aws iot create-policy --policy-name test-thing-policy --policy-document
    '{ "Version": "2012-10-17", "Statement": [{"Action": ["iot:*"],
    "Resource": ["*"], "Effect": "Allow" }] }'
```

```
$ aws iot attach-principal-policy --policy-name temperature-thing-policy
    --principal "arn:aws:iot:eu-west-1:435435435454:cert/aa53893b94c4..."
```

After finishing the setup, you can start monitoring incoming traffic. The example of an AWS console after completing the setup process can be seen in figure 1.

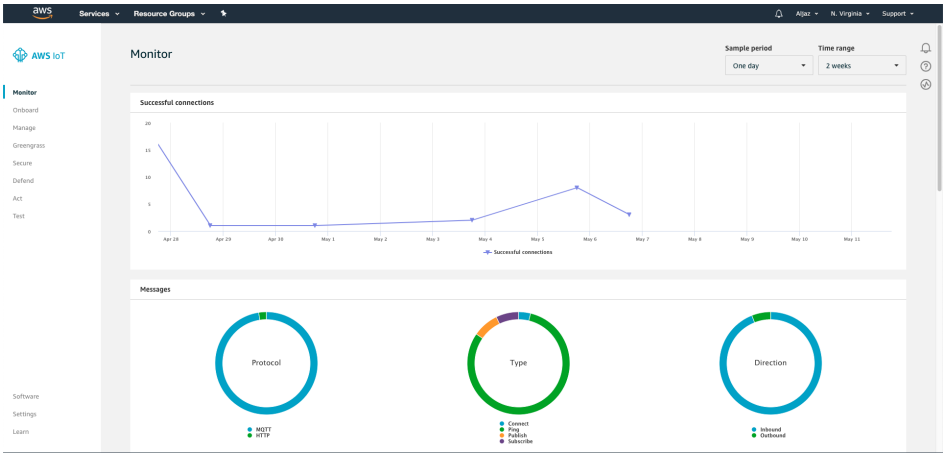

Figure 1. Example of an AWS Console.

CloudMQTT. AWS IoT enforces sensors to authenticate themselves via certificates. The processing, however, takes way too much memory and in our case it causes certain tasks to hang. Thus, we had to find another service which provides MQTT brokers using the Basic authentication i.e. authentication with username and password.

One of them is CloudMQTT, a hosted message broker for the Internet of Things [4]. It is quite easy to set things up, since they have a detailed step by step guide on their website. In the figure 2 we see their dashboard with a set of temperature readings of a test we performed.

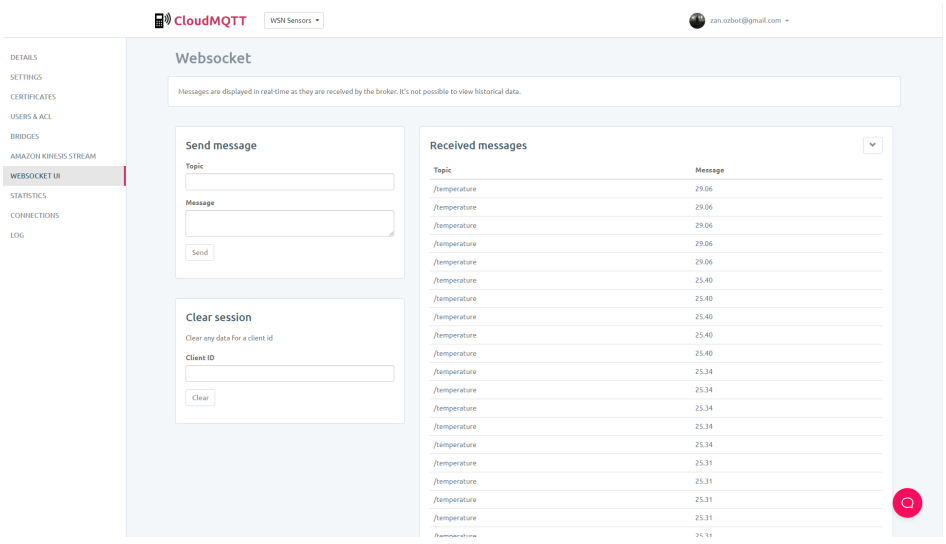

Figure 2. Received messages of a performed test on the CloudMQTT platform.

If you wish to use your own MQTT broker using Basic authentication navigate to the code folder and simply change the following variables. the following variables defined in *mqtt_client.c*.

```
#define MQTT_USER "xxxxxxxxx"
#define MQTT_PASS "XXXXXXXXXXX"
```

2. Code merge and problems

During the whole development process we encountered numerous problems. This section describes the most critical ones.

Merging of the HTTP server and nRF24 module. We came upon our first major problem while trying to join the nRF24 receiver and the HTTP server which are both supposed to run on the master micro-controller. The receiver code and *RF24* library were written in C++, while the HTTP server's *httpd* library was written in C. We decided to use the `extern "C"` option. Unfortunately, the *httpd* library was not compatible with C++. We tried rewriting parts of the library which caused problems to C++. We then noticed, that a large part of the library depended on those parts of the code, so the whole library needed to be rewritten. In the end, we decided not to use the nRF24 module and to collect the temperature on the master controller instead.

Merging of the MQTT SSL and HTTP server. The second major problem was trying to merge the code which used certificates for authentication to work alongside the HTTP server. We tried (i) increasing and decreasing the task's heap size, (ii) fiddling with the task's priorities, (iii) programmatically delaying certain tasks to free up memory, and (iv) merging tasks into one big conglomerate of code. We just could not make them to work together. Either one of them was working or none. As it turns out the processing of certificates which make a secure connection takes just a bit more memory than available, thus making the MQTT task non responsive.

In the process of fixing this issue we had to change the MQTT broker provider. Previous AWS IoT only offered authentication with certificates and by switching to CloudMQTT we managed to get things working using Basic authentication.

Merging of the MQTT, HTTP server and BMP280. The final solution is structured as seen in the figure 3. Each module (i) BMP280, (ii) HTTP server, and (iii) MQTT client is grouped into its own file for better readability. The entry point of the implementation is located in *sensor.c* where all initial configuration is set and from where all the tasks are started.

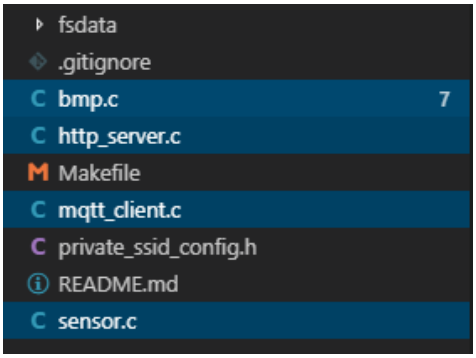

Figure 3. Project structure with main module groups highlighted.

3. Results

After completing the code and testing the implementation, we decided to carry out the temperature measurements in a few different environments.

In the first test, we took the micro-controller from the room temperature (which was about 25°C) and put it in an oven, which was set to 70°C. The results can be seen in the figure 4. As expected, the temperature has risen rapidly at first and then started normalizing. Judging by the graph, the actual temperature of the oven was closer to 80°C, since the temperature was still slowly rising after the end of the experiment at 69°C.

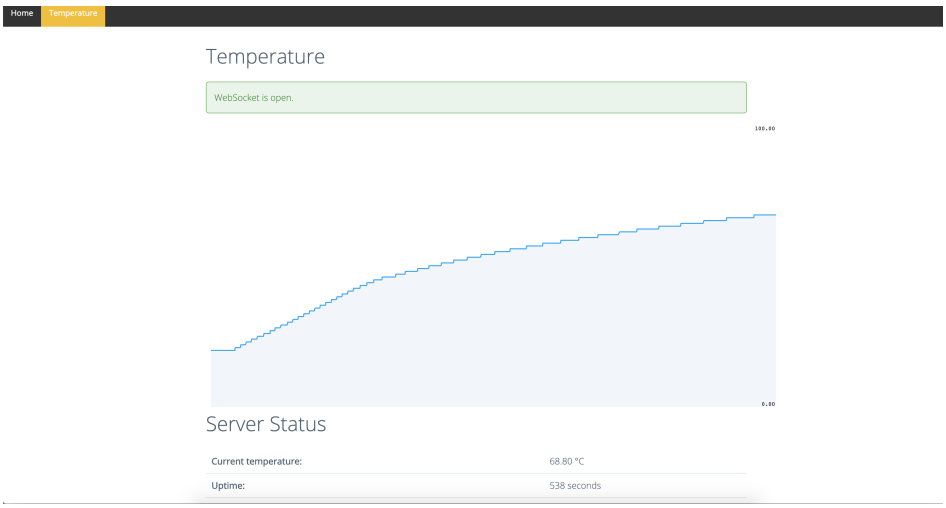

Figure 4. A graph showing temperature increase after putting a micro-controller in an oven.

In the second experiment, we put the micro-controller in front of a fan heater, at a distance of approximately 30cm. As seen in the figure 5, the initial increase of temperature is even more drastic than in the previous experiment. This is the consequence of a fan heater blowing hot air directly in to the micro-controller. The temperature then stops increasing at around 75°C.

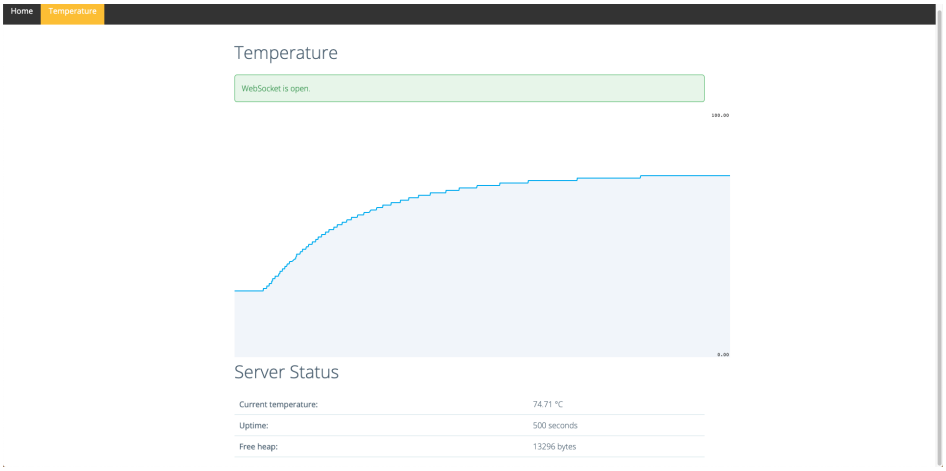

Figure 5. A graph showing temperature increase after putting a micro-controller in front of a fan heater.

We then moved the controller away from the fan heater and let it cool off, which is shown in the figure 6. After ending the experiment, the micro-controller reaches 33°C, which is still higher than room temperature.

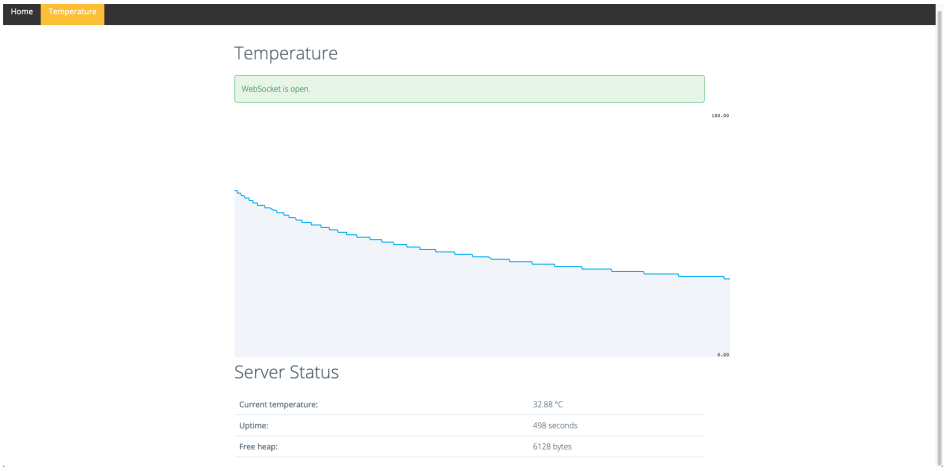

Figure 6. A graph showing temperature decrease after moving the micro-controller away from the fan heater.

The next experiment involved putting the micro-controller to a freezer and taking it out, when it reaches 0°C. As seen in the figure 7, the controller reaches 0°C after around 5 minutes. The controller was then taken out of the freezer to measure the normalization of the temperature to the room temperature. Unfortunately, the browser lost the connection to the WebSocket, which prevented us to measure the whole 10 minute interval. This problem occurred in a lot of experiments, so we had to redo them multiple times.

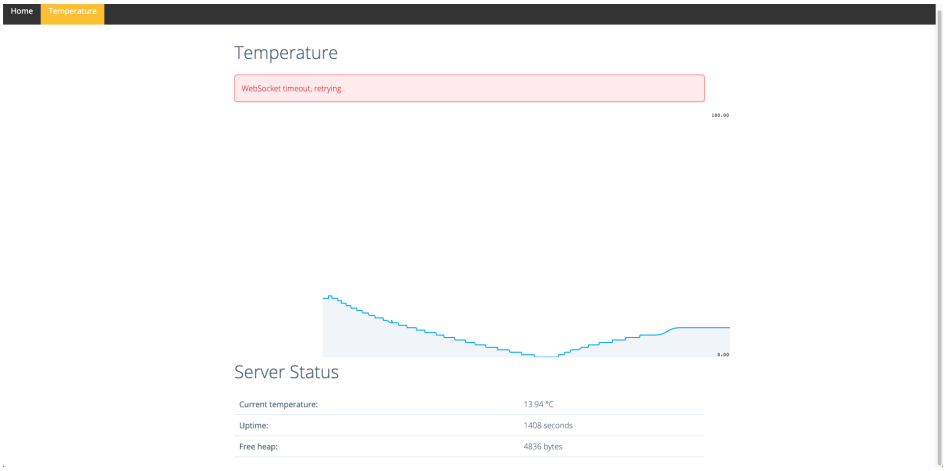

Figure 7. A graph showing temperature changes when putting a micro-controller to a freezer and later taking it out.

In the last experiment, we wanted to find out how hot a persons head gets while using a hair dryer. The subject strapped the micro-controller on top of his head and proceeded to dry his hair in a normal manner. As seen in the figure 8, after about a minute, the temperature on top of the head reaches 50°C. It stays about the same for the next 2 minutes, after which time the subject turns off the hair dryer. The subject then lets the hair cool off while leaving the micro-controller on his head. At the end of the experiment about 5 minutes later, the measured head temperature is around 36°C.

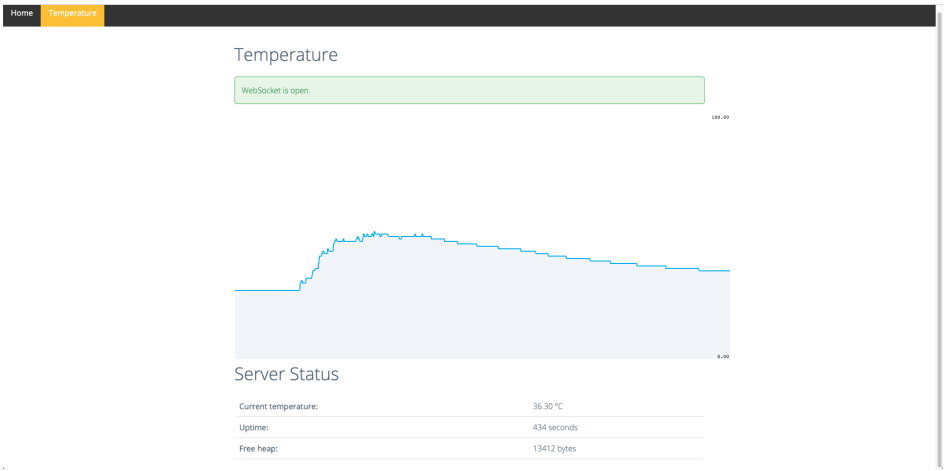

Figure 8. A graph showing temperature changes on subject's head while using a hair dryer.

4. Discussion

Although the final solution differs from the initial goals, due to the problems experienced along the way, we were still able to provide a working one. We programmed our micro-controller to read the temperature using the BMP280 module, send it to a topic using the MQTT protocol and display the live results on a HTTP server.

As for the future work, we intend to decouple the sensor code. We will make only one super node which will display results on a HTTP server and send data to the MQTT broker. Other nodes will only be responsible for reading the temperature data and transmitting it over to the super node using the nRF24 wireless transceiver module. Thus making the network more efficient.

Bibliography

1. (2019) nRF24L01 Product Specification (https://media.digikey.com/pdf/Data%20Sheets/Nordic%20Semiconductor%20PDFs/nRF24L01_PS_July2007.pdf). Accessed online: 11th May 2019.
2. (2019) MQTT (http://mqtt.org/). Accessed online: 11th May 2019.
3. (2019) ECC-based certificates (https://aws.amazon.com/blogs/iot/elliptic-curve-cryptography-and-forward-secrecy-support-in-aws-iot-3). Accessed online: 11th May 2019.
4. (2019) CloudMQTT (https://www.cloudmqtt.com/). Accessed online: 11th May 2019.