Voorwoord

Samenvatting

Table Of Contents

[Voorwoord 2](#_Toc9492930)

[Samenvatting 3](#_Toc9492931)

[Table Of Contents 4](#_Toc9492932)

[Preface 6](#_Toc9492933)

[1 Arduino configuration 7](#_Toc9492934)

[1.1 I²C vs SPI 7](#_Toc9492935)

[1.2 THE SCHEME 8](#_Toc9492936)

[1.3 How does it work? 8](#_Toc9492937)

[1.4 THE CODE: 9](#_Toc9492938)

[1.4.1 MASTER DEVICE 9](#_Toc9492939)

[1.4.1.1 Full master code 12](#_Toc9492940)

[1.4.2 SLAVE DEVICE: 14](#_Toc9492941)

[1.4.2.1 Full slave code 15](#_Toc9492942)

[2 BANK PDU MONITORING 16](#_Toc9492943)

[2.1 Logging in via CLI 16](#_Toc9492944)

[2.1.1 Password recovery 17](#_Toc9492945)

[2.2 Logging in via GUI 19](#_Toc9492946)

[2.3 Configuring SNMP on PDU 20](#_Toc9492947)

[2.3.1 Adding the raspberry pi 20](#_Toc9492948)

[2.3.2 PowerNet MIB 21](#_Toc9492949)

[2.3.3 PDU Firmware upgrade (extra) 23](#_Toc9492950)

[2.4 Installing SNMP on the Raspberry Pi 24](#_Toc9492951)

[2.5 Time issues on the Raspberry Pi 29](#_Toc9492952)

[3 Motion- and smoke sensor 31](#_Toc9492953)

[4 Transport protocol 32](#_Toc9492954)

[4.1 MQTT vs AMQP 32](#_Toc9492955)

[4.1.1 MQTT 32](#_Toc9492956)

[4.1.2 AMQP 33](#_Toc9492957)

[4.2 IMPLEMENTING MQTT 34](#_Toc9492958)

[4.3 TLS 35](#_Toc9492959)

[4.4 IMPLEMENTING TLS 36](#_Toc9492960)

[4.4.1 Certificate Authority 36](#_Toc9492961)

[4.4.2 Server Certificate 38](#_Toc9492962)

[4.4.3 Client Certificate 39](#_Toc9492963)

[4.4.4 Mosquitto.conf file 40](#_Toc9492964)

[4.5 Example on the command line: 41](#_Toc9492965)

[4.6 Example from program code: 42](#_Toc9492966)

[5 RASPBERRY PI 43](#_Toc9492967)

[5.1 Logging in 43](#_Toc9492968)

[5.2 The Code 45](#_Toc9492969)

[6 testing the connection 51](#_Toc9492970)

[7 AUTOSTART SCRIPT AT BOOT 52](#_Toc9492971)

[8 payload encryption (extra) 54](#_Toc9492972)

[9 1-wire sensors on the Pi (extra) 55](#_Toc9492973)

[9.1 The scheme 56](#_Toc9492974)

[9.2 The code: 57](#_Toc9492975)

[Conclusion…. 59](#_Toc9492976)

[Bibliography 60](#_Toc9492977)

Preface

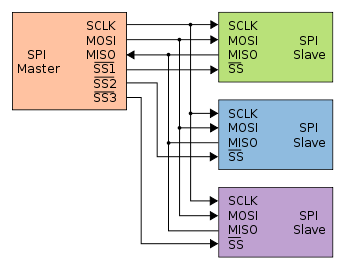
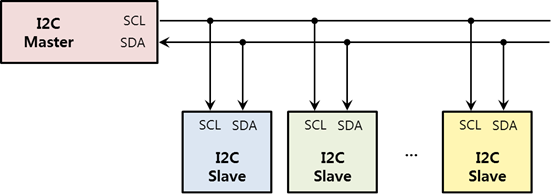
The goal of the project is to create a monitoring system suited for the datacenter in the University of Jimma. We will create the entire framework that will be installed in the datacenter. In the future this framework can be reused for other projects. For this project several devices must be monitored. We will use temperature and humidity sensors that will be installed on 3 of the racks in the datacenter. The racks are powered by Metered PDU Racks. It is possible to retrieve information from these PDU’s using SNMP. Finally, we will also install a smoke- and motion sensor. All this information will be gathered by several devices and made available for the API to collect this data and put it in the database. Afterword’s the data will be visualized in a dashboard which the staff can use to monitor the devices in the datacenter.

# Arduino configuration

## I²C vs SPI

Because we want to monitor the datacenter at different places, we should best use a bus protocol. This way we can connect multiple devices. One device will be our master device, controlling the other slave devices. The sensors will be connected to the slave devices. When using Arduino’s or Raspberry Pi’s both the SPI and I²C protocol can be used to connect the devices in a bus. The following pictures help explaining the two protocols.

SPI: I²C:

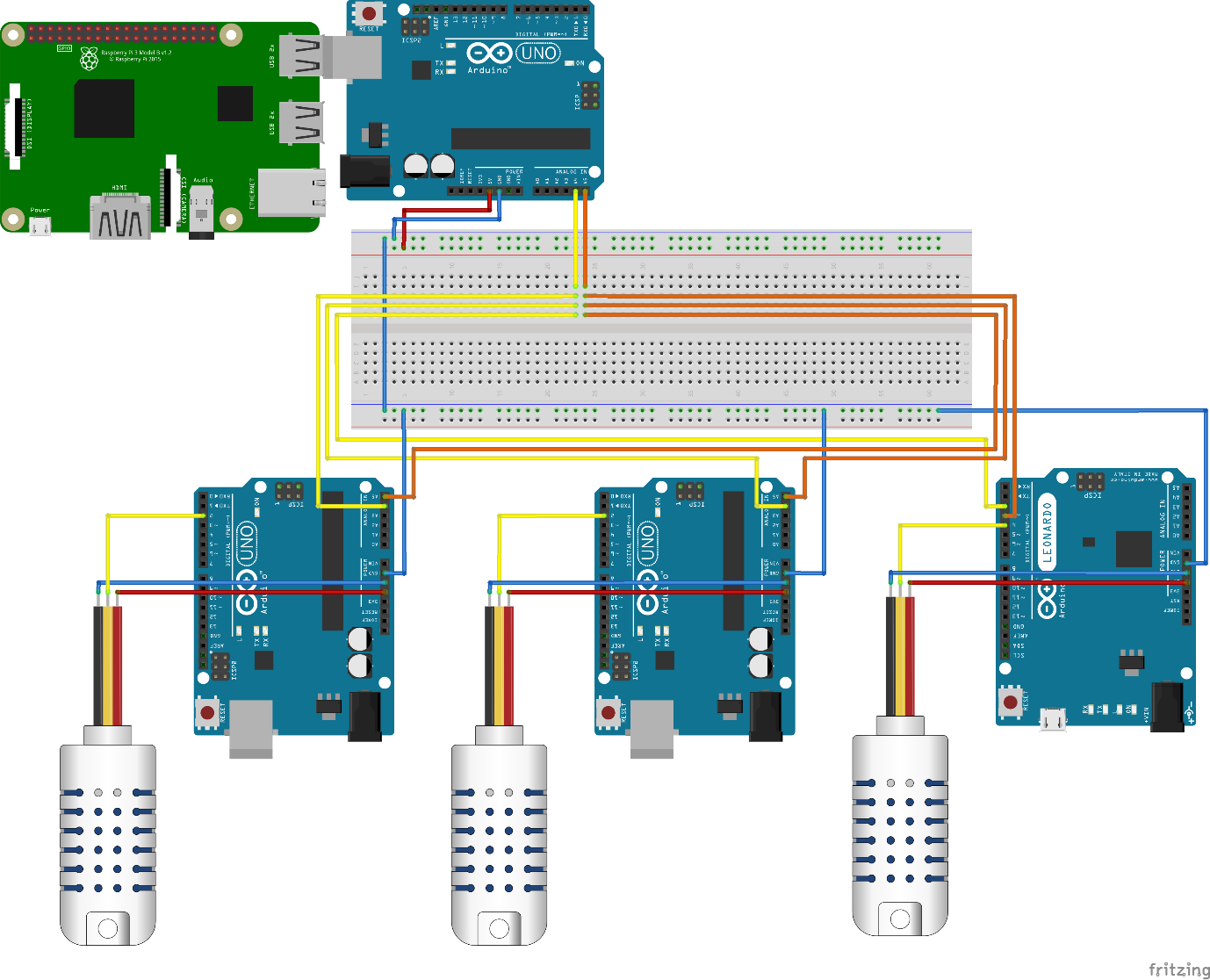


As you can see on the pictures the biggest difference between the two protocols is the wiring. I²C only needs two wires for communicating, one clock- and one data wire. All the devices also should have a common ground. This is not displayed on the photo. SPI needs four wires. For each slave that you want to connect to the SPI bus you will need a different SS (Slave Select or Chip Select) line which makes more complex wiring. With this line you tell the slave that you want to send or receive data by pulling it low. De data that is being send goes through the MISO (Master In Slave Out) or MOSI (Master Out Slave In) lines, a big advantage for SPI is that it can send data full duplex meaning that data can be send simultaneously in both directions. I²C doesn’t support this feature but in our configuration that also won’t be necessary. Another advantage of SPI is that it can transmit data at higher speeds then I²C, but it can only do that for short distances while I²C on the other hand can transmit data over greater distances. Some other advantages of I²C over SPI include: less susceptible to noise, cheaper to implement and the guarantee that the transmitted data is received by the slave.

It is also possible to use a Raspberry pi as master device with the SPI protocol. With this configuration it would be possible to connect more than one slave device on the bus because the Pi does have more than one chip select pin. But when using a Raspberry pi, you should also implement a logic level converter to not damage the pins on the Pi. Since we don’t have that kind of device at our exposal, we will use Arduino’s. We also prefer Arduino’s over a Raspberry Pi because using a Raspberry Pi in a configuration as this would be overkill since we only want to perform repetitive tasks like control several devices and gather data. The Pi has much more resources and consumes more power making it not suitable just for these tasks. Later, we will use the Pi to act as a gateway between our devices gathering the data and supplying this data to the database.

For the project we choose the I²C protocol because there is need of less wiring and when you want to use an Arduino as master in SPI you can only connect one slave device since there is only one chip select pin available on the Arduino

## THE SCHEME



## How does it work?

On the top of the image you can see the Arduino master. He will send a message to each slave (the Arduino’s at the bottom of the image) requesting data. In turn each slave will respond with the data it has collected from the [temperature- and humidity sensor](https://cdn-shop.adafruit.com/datasheets/Digital+humidity+and+temperature+sensor+AM2302.pdf) (the device on the left of each slave). After the last slave sends the data to the master it will start again by requesting data from the first slave. This loop will continue to run until you shut down the master device. The yellow line coming from the master is the data line (SDA), we can connect each data line from the slave devices to this line. The requested data will be sent via this line in the form of bytes. The orange line is the clock line (SCL). This line is used for synchronization of the data transfers over the I²C bus. When using a I²C bus you need to make sure that all the devices are connected to the same ground, so they have the same reference point. The data that is collected by the master will immediately be send through to the Raspberry pi via the Serial interface. When the data arrives at the pi it will be decomposed and put into variables ready to be transmitted to the database.

If one of the slaves gets disconnected from the bus zeros will be displayed as values meaning there is some kind of issue. If one of the devices isn’t supported anymore by power the bus will jam. Then you will have to physically check which of devises isn’t receiving any more power. The bus will jam because it functions in open drain. This means it is always high and when there is communication it will be pulled low by a device. When a device doesn’t have any power, it will pull the bus low. Therefore, the bus will endlessly wait for the device to become back online. When power is reinstalled the bus starts working again. It is possible that this will generate an error at the pi because he may have received some previous values and has now to many values send towards him. You can check this if the master device led is blinking but the data isn’t coming through on the dashboard. Normally the Pi is configured to restart the program itself when it should fail. When this doesn’t happen, you must reset the program on the pi or reset the pi.

## THE CODE:

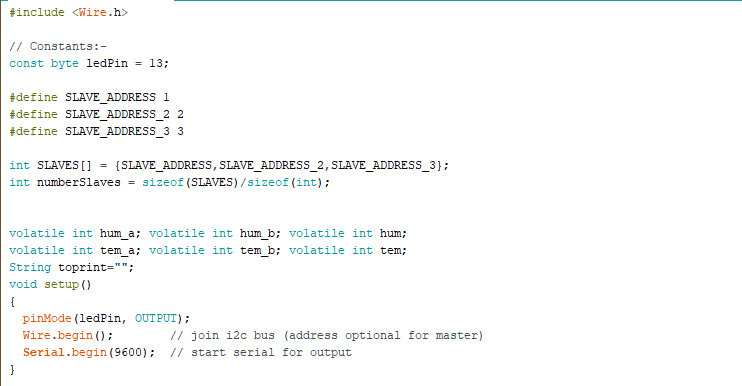
### MASTER DEVICE

We start our code by including the Wire.h library. This library allows you to communicate with I²C devices. Afterword’s we declare a pin which will toggle when data is being transmitted.

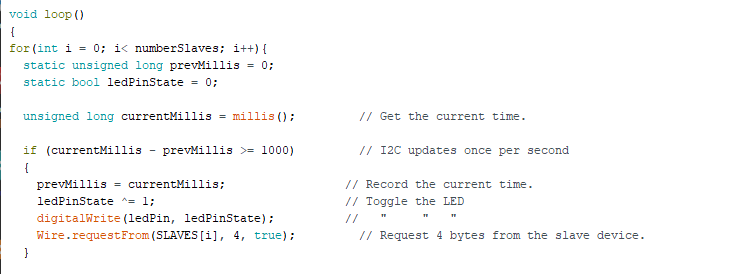
Next, we declare the addresses of the slave devices connected on the bus. Then we put them in an array which we will use later to perform the requests to all the slaves. If you add another slave on the bus you can just add a name and give it an address, then add it in the array. In C code u can get the size of an array in bytes using sizeoff(array), but to get the number of elements you divide this by the size of a single element. That’s why we make use of the division in the part to get the number of slaves.

In the last declaration part, we set the variables where we will store our data that we received from the slaves. We choose volatile as an extra keyword because the value of these variables can be changed beyond the scope of this declaration.

After the declaration part we prepare the Arduino to run the rest of its program. This part will only run once. First the led pin is initiated as an output pin. Secondly, we use the command Wire.begin() to join the I²C bus and finally we initiate the Serial interface and set the baud rate at 9600.

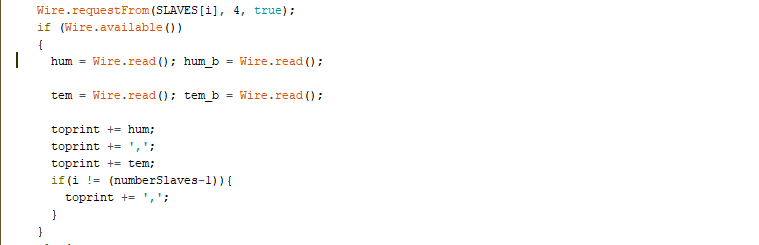


Next, we write the main program that will be repeated constantly. We use a for loop that will repeat for the number of slaves that or connected on the bus. In the beginning we declare some variables that are necessary to synchronize the bus. This synchronization is necessary if we want to read the data from the bus at the correct time.



After the synchronization we start with a request for data from our first slave. In the Wire.requestFrom() function we enter the address of the slave, the number of bytes we expect to receive and that he should release the data line once he has sent his data, this is done by putting a Boolean of true as final parameter. We want to read the 4 bytes that is send by the slave. If we add a sensor to the slave, we must change this parameter to 6 or more depending on how many values we want to send from the slave.

Next, we wait for a response of the slave, if the slave is on the bus, we enter our if-statement. Each Wire.read() function reads out 1 byte that is send by the slave. Each byte that is send represents a number. The first byte represents the humidity the third byte represent the temperature. For some reason the second and fourth byte return zero and contain none of the data that is transmitted. But we need to read out these bytes since Wire.read() reads out the bytes in numerical order. I couldn’t find a way to just send two bytes containing the 2 values. With I²C the first byte that is being send contains the address of the destination. This could be the reason why de bytes appear on other positions then intended and why it is not possible to send just 2 bytes.

Next, we create a string that holds both values separated by a comma. When we receive the last value there should no comma be placed after. 

If the device is not available, f.e. the wires are disconnected, we will put zero’s in the variables. The rest of the program is the same as in the if-statement. Because we still fill the values of the variables, we can check which of the slaves is disconnected. By printing it out via the serial command the pi will also receive these values since he is connected via the serial interface. This code is performed for each slave.



In the example we can see that the wires of the first slave have been disconnected. For some reason the temperature of the second slave also returns zero, probably because the bus picks up a wrong signal from the disconnected slave, given it some sort of synchronization problem.



#### Full master code

#include <Wire.h>

// Constants:-

const byte ledPin = 13;

#define SLAVE\_ADDRESS 1

#define SLAVE\_ADDRESS\_2 2

#define SLAVE\_ADDRESS\_3 3

int SLAVES[] = {SLAVE\_ADDRESS,SLAVE\_ADDRESS\_2,SLAVE\_ADDRESS\_3};

int numberSlaves = sizeof(SLAVES)/sizeof(int);

volatile int hum\_a; volatile int hum\_b; volatile int hum;

volatile int tem\_a; volatile int tem\_b; volatile int tem;

String toprint="";

void setup()

{

pinMode(ledPin, OUTPUT);

Wire.begin(); // join i2c bus (address optional for master)

Serial.begin(9600); // start serial for output

}

void loop()

{

for(int i = 0; i< numberSlaves; i++){

static unsigned long prevMillis = 0;

static bool ledPinState = 0;

unsigned long currentMillis = millis(); // Get the current time.

if (currentMillis - prevMillis >= 1000) // I2C updates once per second

{

prevMillis = currentMillis; // Record the current time.

ledPinState ^= 1; // Toggle the LED

digitalWrite(ledPin, ledPinState); // " " "

Wire.requestFrom(SLAVES[i], 4, true); // Request 4 bytes from the slave device.

}

Wire.requestFrom(SLAVES[i], 4, true);

if (Wire.available())

{

hum = Wire.read(); hum\_b = Wire.read();

tem = Wire.read(); tem\_b = Wire.read();

toprint += hum;

toprint += ',';

toprint += tem;

if(i != (numberSlaves-1)){

toprint += ',';

}

}

else{

hum = 0;

tem = 0;

toprint += hum;

toprint += ',';

toprint += tem;

if(i != (numberSlaves-1)){

toprint += ',';

}

continue;

}

delay(1000);

}

Serial.println(toprint);

toprint="";

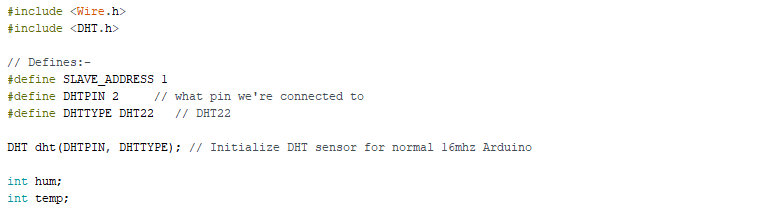
}

### SLAVE DEVICE:

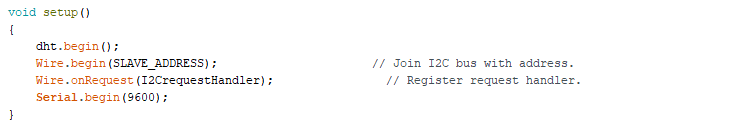
Just like with the master we start our code by including the Wire.h library. This library allows you to communicate with I²C devices. Apart from the Wire.h library we also need to include the DHT.h library. This library gives us the possibility to read out the connected DHT-sensor. The DHT-sensor is a digital, one-wire sensor.

Next we declare the address of the slave device connected on the bus and tell the Arduino on which pin the DHT-sensor is connected to. The DHT-sensor can detect temperature and moisture.

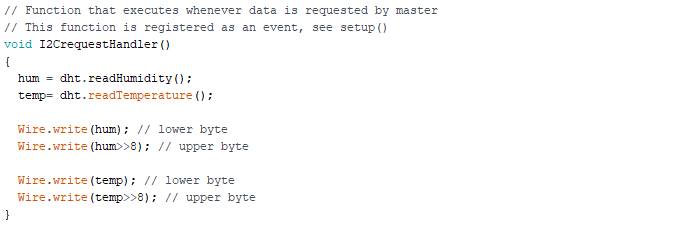
Finally we declare to variables where we can store in the data from the sensor.



In the setup part we start our sensor and join the I²C bus with our slave. Then we tell the device to listen to a request coming from the master. In this function we put the function to fulfill when a request is present. The last part is setting the baud rate of the Serial connection. This can always be useful for debugging when opening the serial monitor. This way we can read out a sensor and Serial.print() the value so we can check if the sensor is working correctly.



The final part of the code consists of the requestHandler function. As said before this function will executes whenever data is requested by the master. We simply read out the data from the sensor and put that data into the 2 variables. This data then needs to put on the data line of the I²C bus. If you add a sensor to the slave, you will have to read out its value (f.e. AnalogRead(A0)) and put it into a new variable. Next you can put the data on the data line the same way as before with Wire.write().



#### Full slave code

#include <Wire.h>

#include <DHT.h>

// Defines:-

#define SLAVE\_ADDRESS 1

#define DHTPIN 2 // what pin we're connected to

#define DHTTYPE DHT22 // DHT22

DHT dht(DHTPIN, DHTTYPE); // Initialize DHT sensor for normal 16mhz Arduino

int hum;

int temp;

void setup()

{

dht.begin();

Wire.begin(SLAVE\_ADDRESS); // Join I2C bus with address.

Wire.onRequest(I2CrequestHandler); // Register request handler.

Serial.begin(9600);

}

void loop()

{

delay(500);

}

// Function that executes whenever data is requested by master

// This function is registered as an event, see setup()

void I2CrequestHandler()

{

hum = dht.readHumidity();

temp= dht.readTemperature();

Wire.write(hum); // lower byte

Wire.write(hum>>8); // upper byte

Wire.write(temp); // lower byte

Wire.write(temp>>8); // upper byte

}

# BANK PDU MONITORING

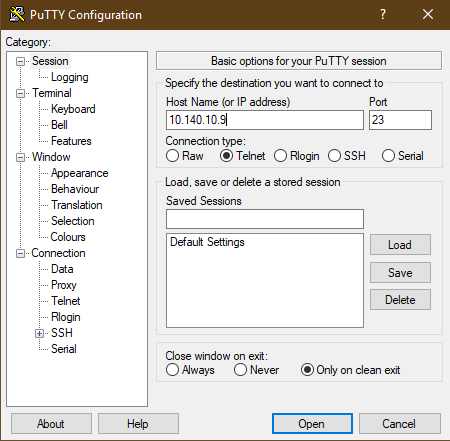
We also want to measure the Rack PDU’s to which the switches are connected to. This way we can monitor the power consumption of the rack as well as the load status of the banks on the rack. PDU level monitoring is useful to determine whether no phase / bank is being overloaded so that certain equipment is no longer fed redundantly. A useful user guide for the APC is found online. We are using the AP7853.

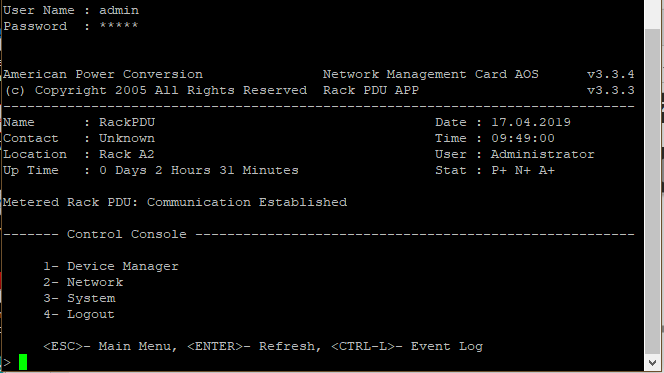
## Logging in via CLI

First we need to connect the Rack PDU to the network. When the PDU receives it’s ip address we can login to it using a terminal program (e.g. putty). Fill in the correct ip address of the PDU and connect to it using Telnet. A terminal will appear where you can fill in the username and password, default this is apc, apc. I changed this to admin, admin.

You can change the credentials both with CLI as with the GUI. The addresses of the PDU’s are:

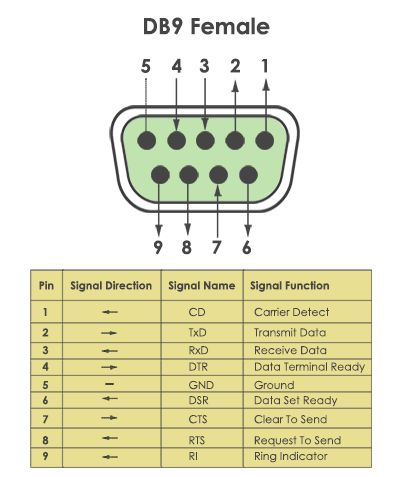
* 10.140.4.151
* 10.140.4.152
* 10.140.4.153





### Password recovery

I encountered some problems logging in since the PDU has been used before and there was a password configured on it. I first had to do a password recovery. Therefore, I needed the correct cable, being a RJ12 to db9(serial) cable. Since there was no cable available, I made this one myself with the help of a staff member of the university. The tools you need are a RJ12 cable, a db9 connector, a serial to USB convertor and a pair of scissors.

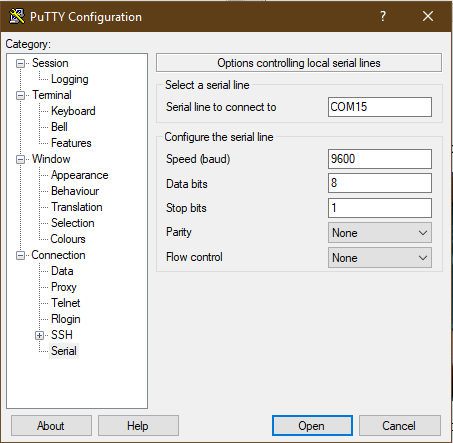
You will need to connect the wires accordingly:

RJ12 2 – black -----> DB9 5

RJ12 3 – red -----> DB9 2

RJ12 4 – green -----> DB9 3

Once you have the cable plug it into the Serial port.

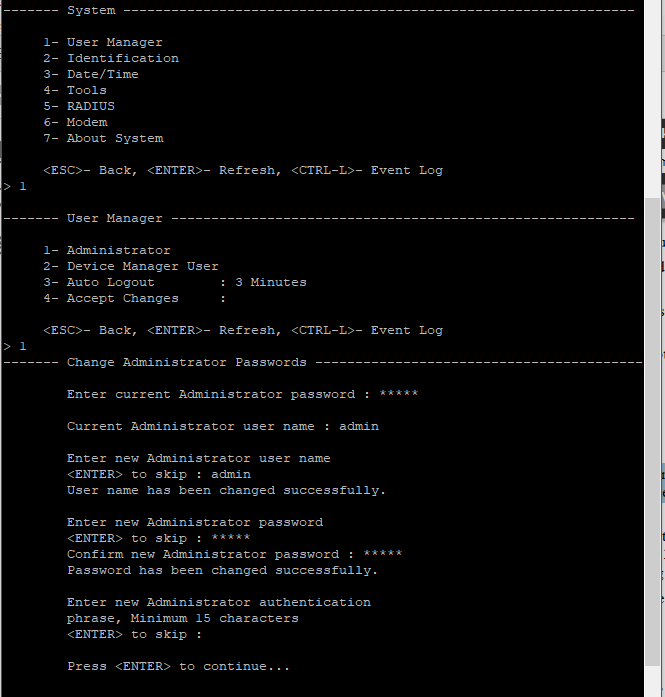
Start up a terminal program and configure the port using the following parameters:



U need to find the correct port to which the PDU is connected. Locate this port in Device manager on your pc. It is possible that you need to install a driver for the USB to Serial convertor.

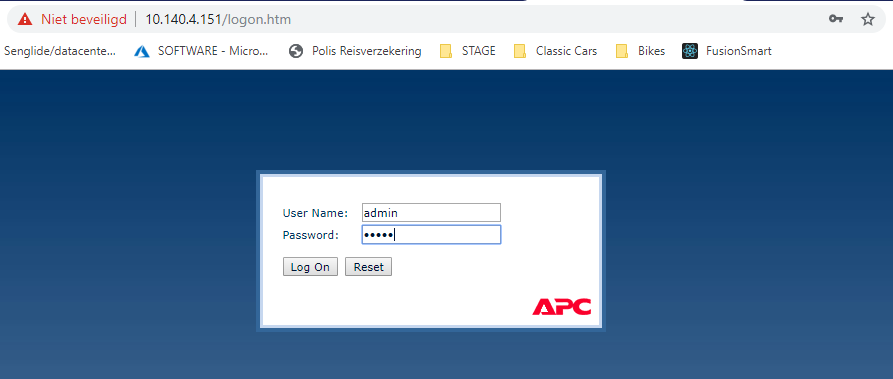
Once you open the Serial connection press Enter until you see the user prompt appears. Then Press the Reset button. The Status LED will flash alternately orange and green. Press the Reset button a second time immediately while the LED is flashing to reset the username and password to their defaults temporarily.

Press enter again to display the User prompt. Then use the default, apc, for username and password. You should be able to login. Once you entered the console you can find your way to the accounts and change the username and password to whatever you choose.

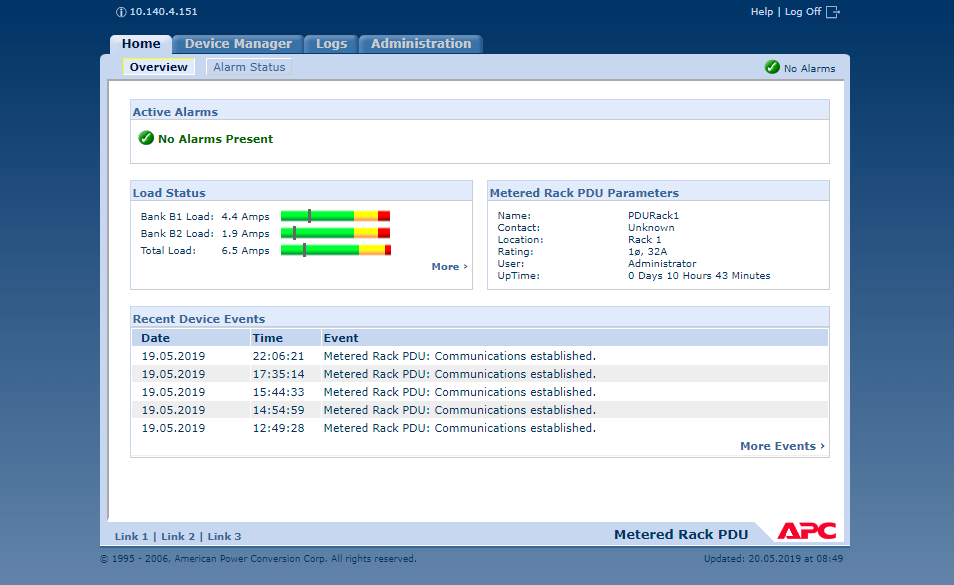


## Logging in via GUI

You can also login using the web interface. Go to the ip address of the PDU and enter the correct credentials (admin admin).



Here you can see the basic information of the rack. it is also possible to configure alarms, change network settings, manage users and many more.

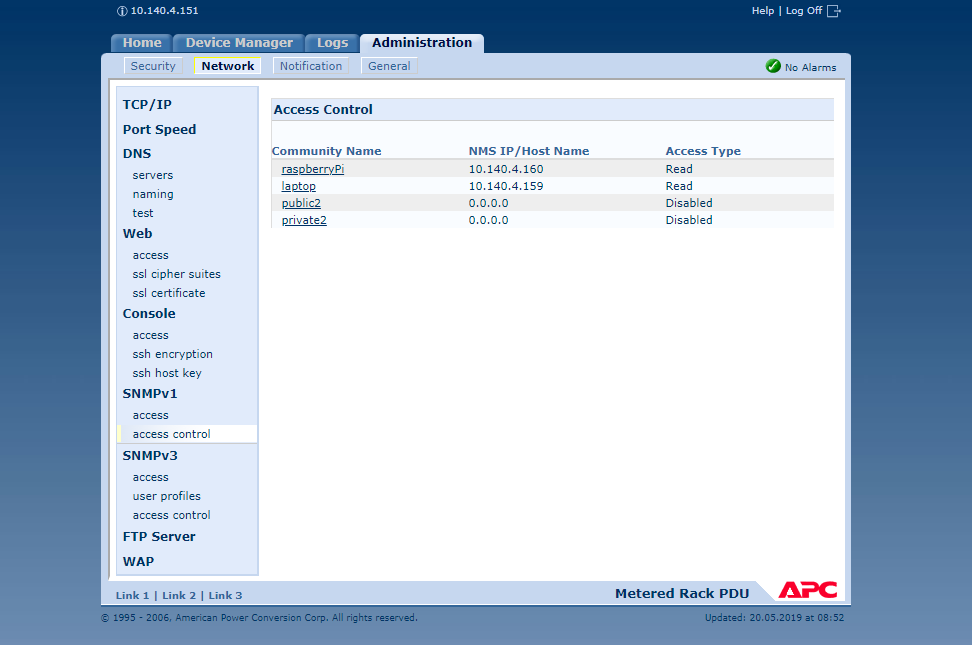


## Configuring SNMP on PDU

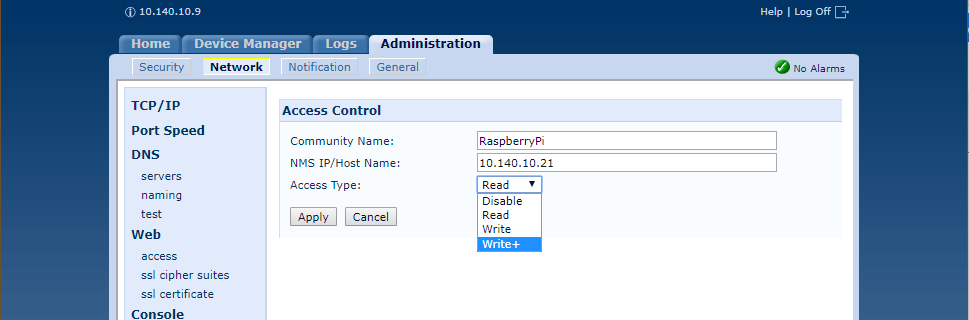
You can configure SNMP both with CLI as with GUI. In the next steps I will show how to do this using the GUI.

### Adding the raspberry pi

In order to make it possible for the Raspberry pi to perform queries on the PDU we need to configure the PDU to allow read access from the pi. In order to do this, go to the administration tab and select the Network tab. Here you see on the left side a list with possible network features to configure. Go to the SNMP – access control field. Here you can add the address of the device. For our project the pi only needs read access, all the other community names can be disabled. By adding the ip address of the pi to the list it will be possible to send SNMP queries.

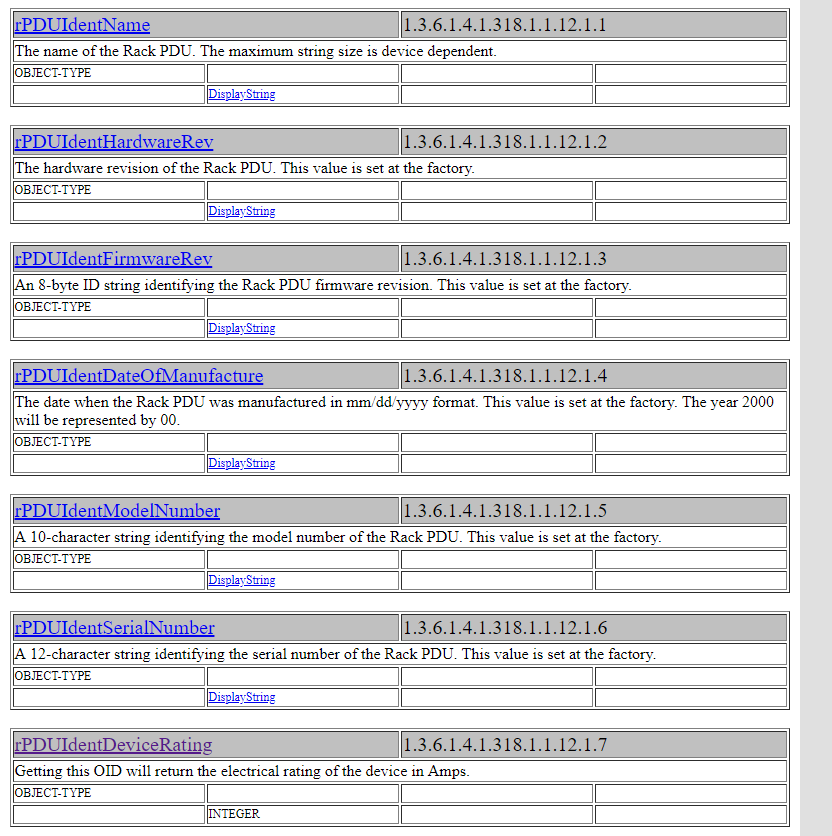


If you click on a community name you can change all the options.



### PowerNet MIB

We want to gather information regarding the power consumption and load status of the PDU. The PDU is configured with the use of SNMP. SNMP is a protocol which you can use to send and receive data from a device using OID’s. OID’s are identifiers and determine a path to a certain value holding information of a part of the device. We can use the PowerNet MIB as a library to search for all the OID’s that are of value to us. Here is a part of the library:



To test whether an OID is available for our device I downloaded a program called SNMP-tester which send queries to the PDU requesting information. First you need to enter your ip address and the ip address of the device you want to send the SNMP requests to, in our case the PDU. Then you can fill in an OID or a path to several OID’s. After some testing I realized that the useful IOD’s where situated in the path 1.3.6.1.4.1.318.1.1.12.

{iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) apc(318) products(1) hardware(1) rPDU(12)}

For full explanation of all the IOD’s I suggest you read the section in the MIB of APC.

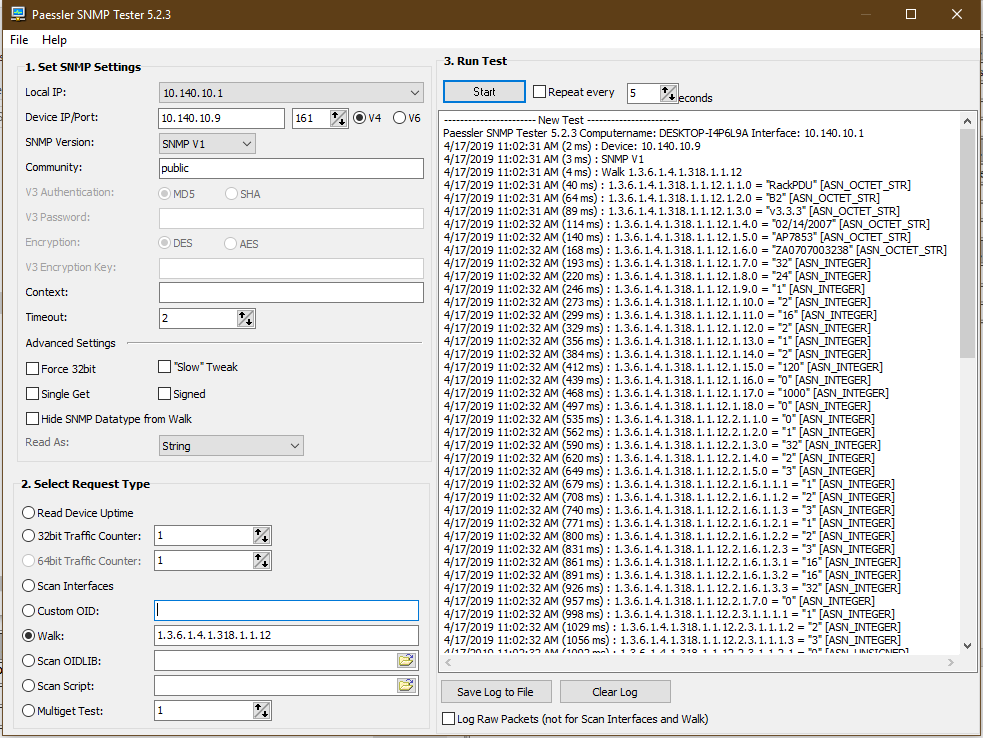
<http://www.circitor.fr/Mibs/Html/P/PowerNet-MIB.php>

We are working with old PDU’s (AP7853) and therefore a lot of the IOD’s aren’t available. For this project we are interested in the following:

devicePowerwatts 1.3.6.1.4.1.318.1.1.12.1.16.0

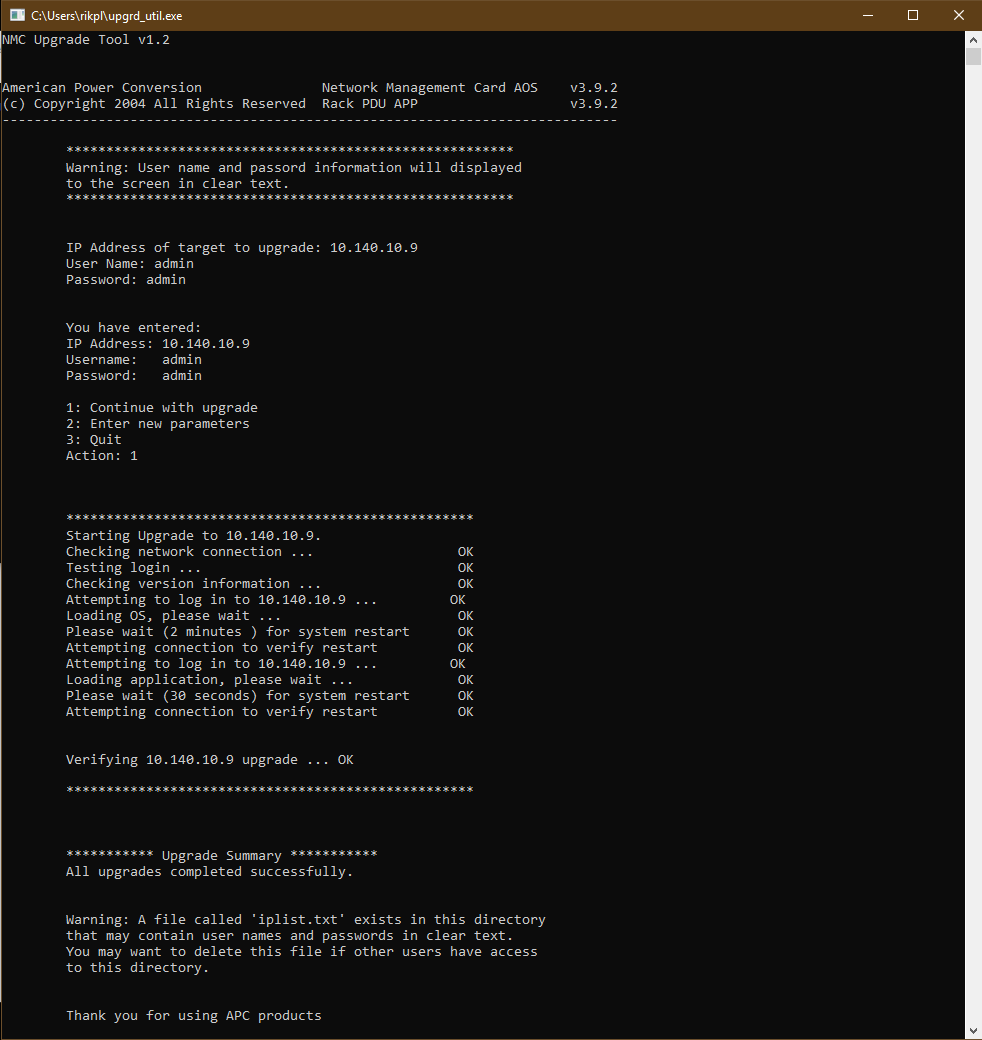
rPduStatusload 1.3.6.1.4.1.318.1.1.12.2.3.1.1.2/ .1 / .2 / .3

the 1,2 and 3 stands for the first bank, the second bank and the total bank



### PDU Firmware upgrade (extra)

You can also perform a firmware upgrade on the APC. Download from the website of APC and give in the correct version of your equipment. You can then download an executable file. Connect a computer to the same network as the PDU. When you open this file, the following appears. Fill in the credentials of the PDU.



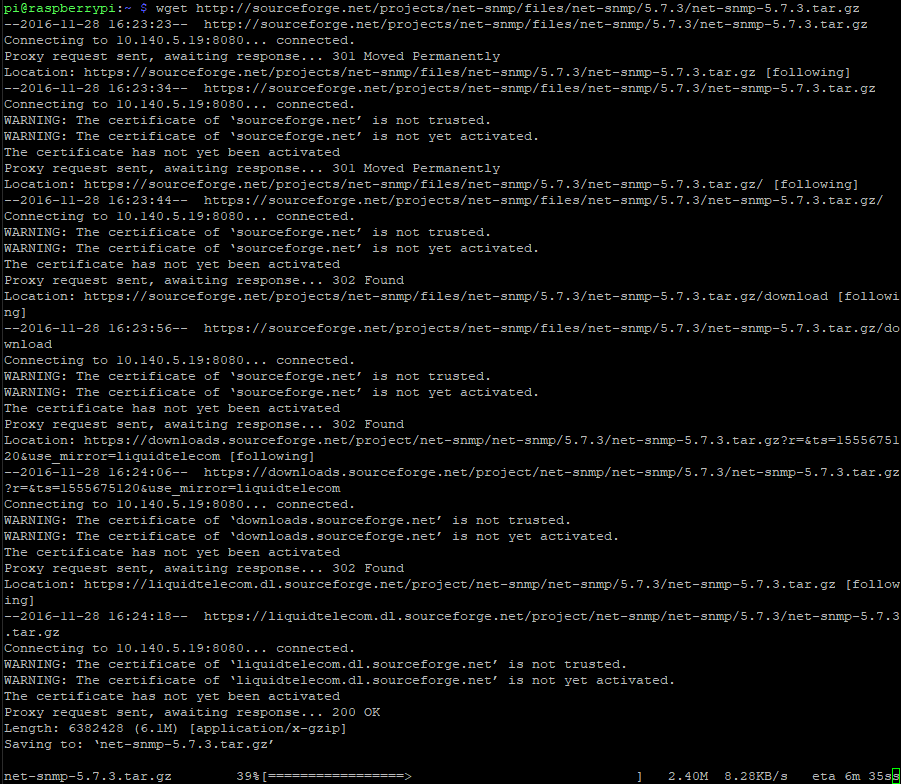
## Installing SNMP on the Raspberry Pi

First you will need to download net-SNMP using:



I used the no-check because the certificate from soundforge isn’t valid anymore. But our SNMP client will only work with this version of net-SNMP, so we work around the certificates. If the –no-check-certificates option is not working wget didn’t install with all options included. You can fix this by making a file called: .wgetrc. and put this in the home directory. In the file write: check\_certificates = off.

Once we enter the command, we will encounter some errors because of the certificates problem but in the end the package should start downloading and you will see something like this:



Next, we need to install the Perl library because net-SNMP has dependencies to this package:



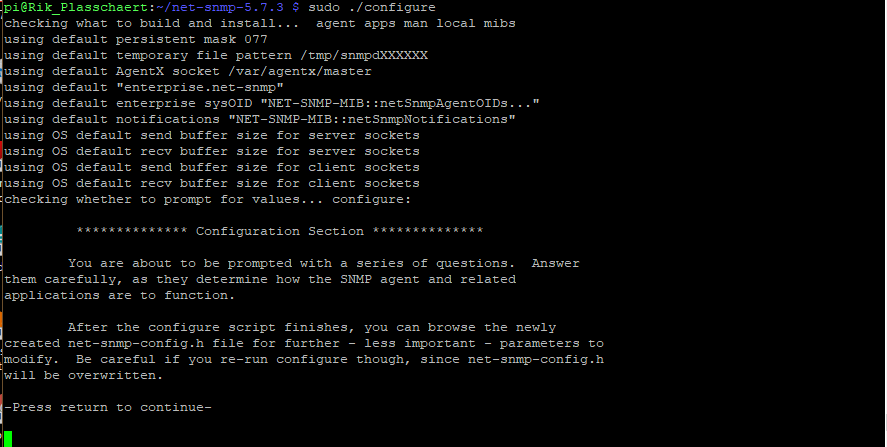
After this is done, we can extract the downloaded file using:

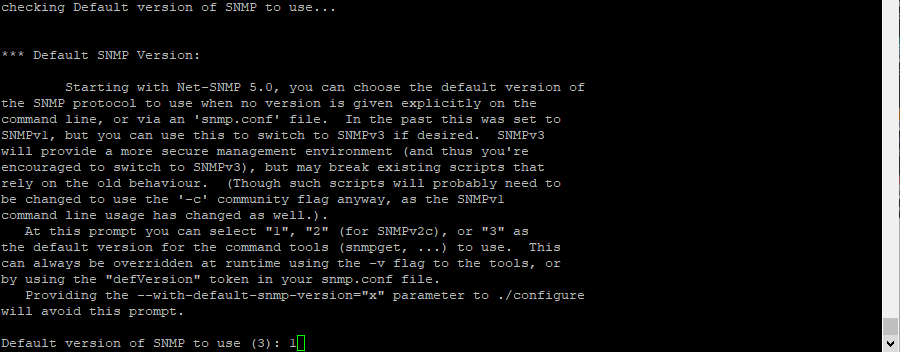


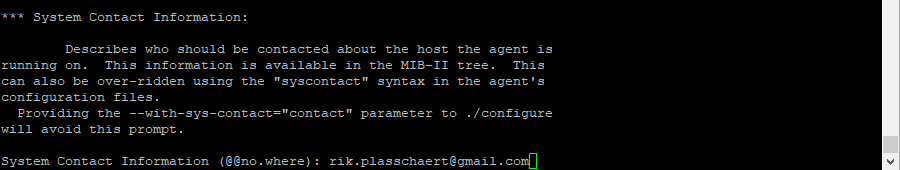
then we enter the main directory and configure the package:

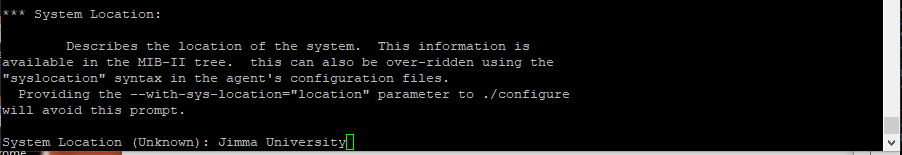


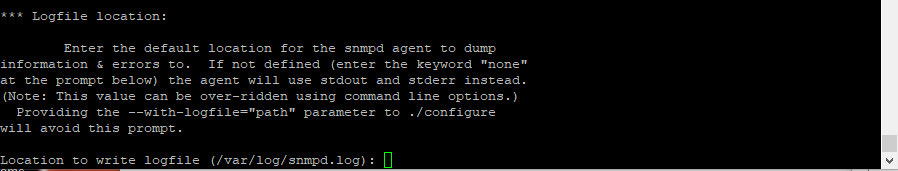
here we will need to fill in a few queries:

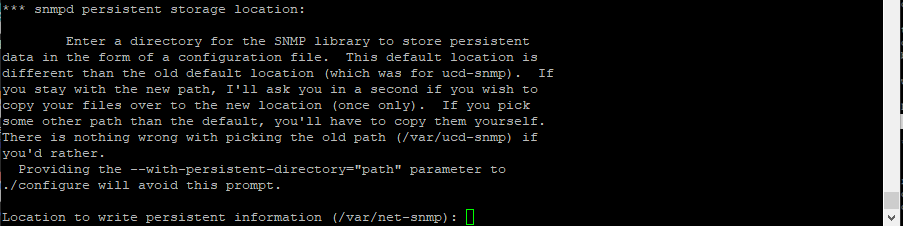


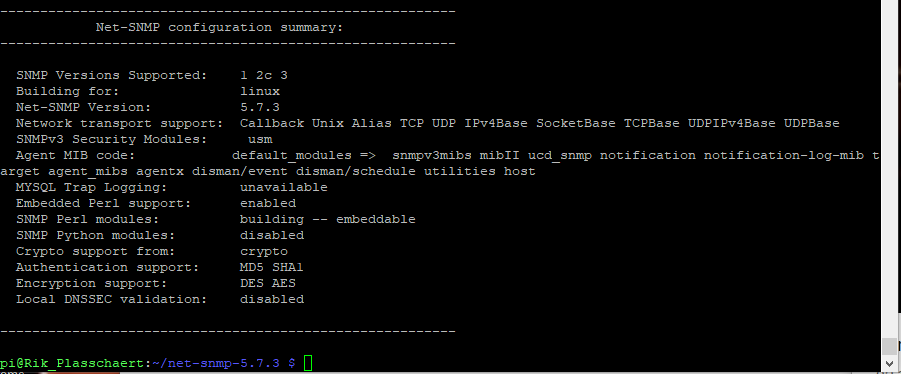










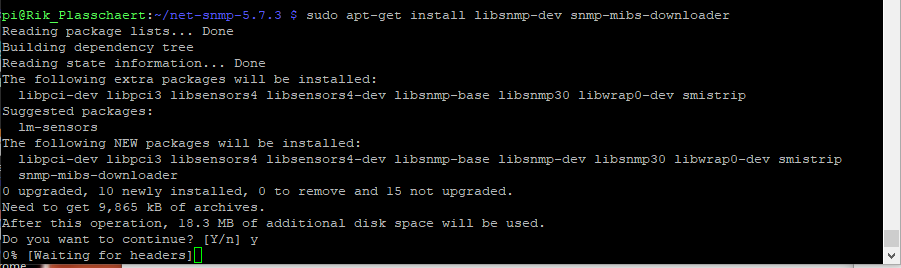
In the end you should see a similar summary. For the moment python modules are disabled, we will enable those in one of the next steps.

This means that the configuration part is successful. Next up, compile and install the package using the following two commands:

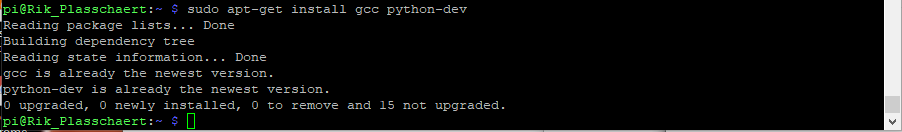




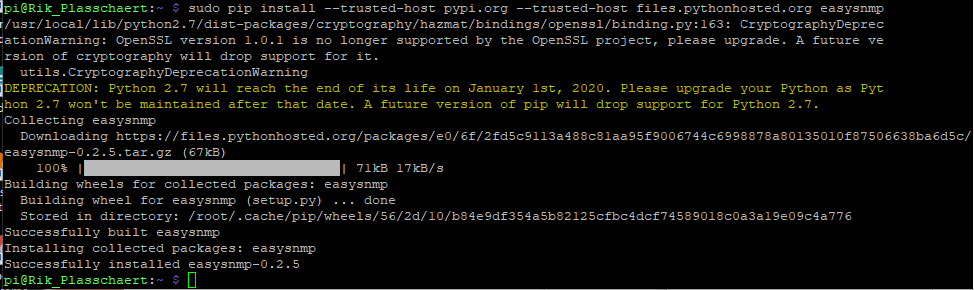
After this we install the easy SNMP library and we download and install the MIB on the system:



Next install the following python packages if not already installed:



Finally, we install easysnmp:



Because there were problems with the SSL certificate, we added the trusted host parameter.

I encountered some difficulties trying to upgrade pip, or any other program that needs to be installed using pip. In the end this command worked for me: 

sudo -H pip install --trusted-host pypi.org --trusted-host pypi.python.org --trusted-host files.pythonhosted.org --upgrade pip

So, if you wish to install or upgrade a program and there are errors regarding certificate verification u can use the trusted-host option. Only use this option if you are sure if the host you want to connect to!

Also install easy-SNMP for python3:



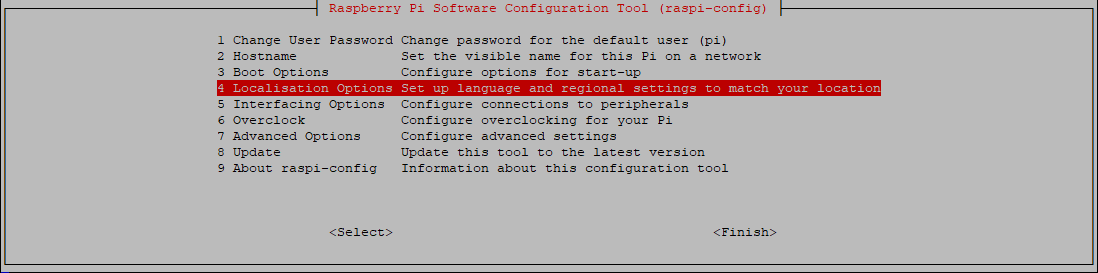
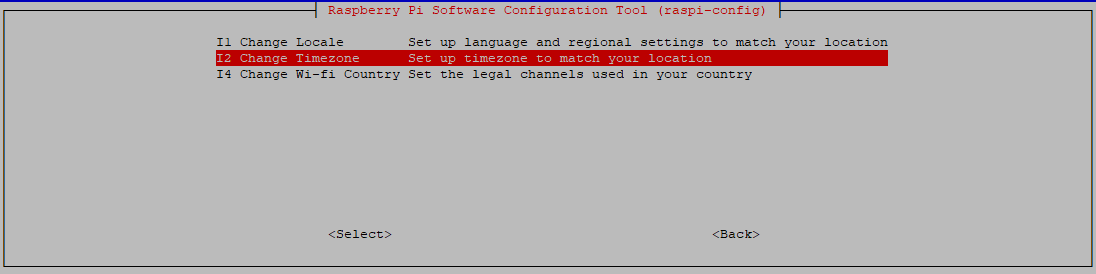
## Time issues on the Raspberry Pi

I could be that the errors occurred because the System time of the pi was wrong.

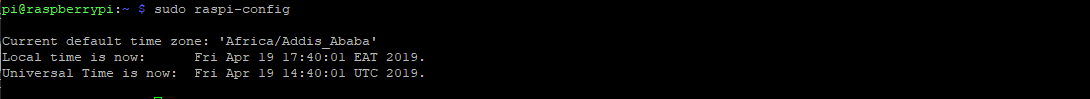


To change the system time, use the following command with your correct time.

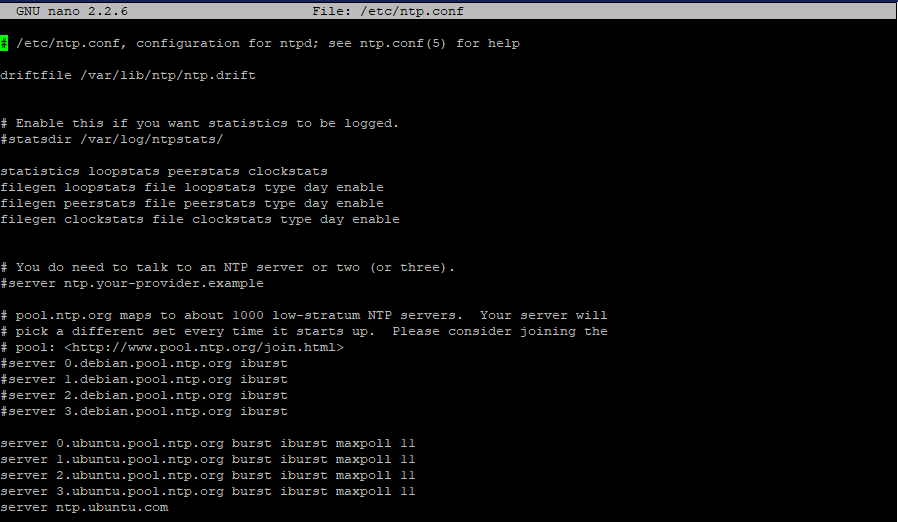


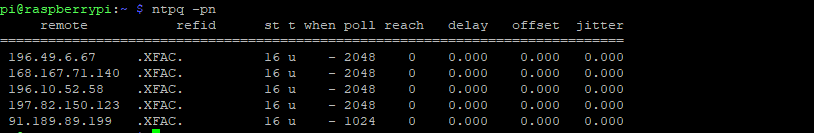
Before you do so make sure that the Pi’s time zone is correct. This project was completed in Jimma, Ethiopia. Enter sudo raspi-config on command line. Then choose next option:  

After you choose your location the following should appear:

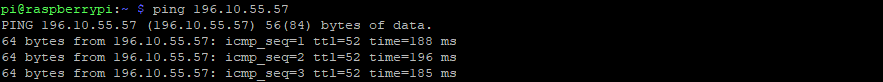


If the Pi is offline for a long period of time he will not be able to correc it to the correct date/time. There is a problem in the network that causes the Pi not being able to connect to the ntp-servers that are listed in the config file.

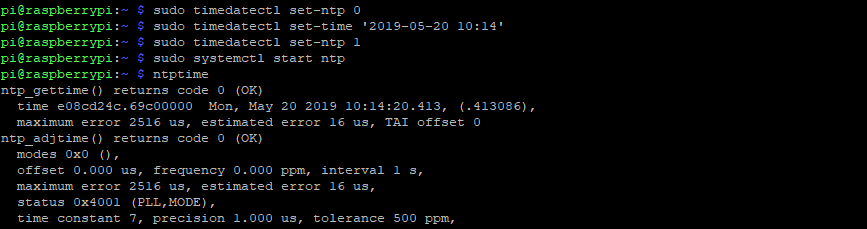




Even though it is possible to ping the ntp-servers.



With the following commands the timesync will be working again for some time but as long as the pi can’t make a connection to the ntp-servers the time will start to tick off again.



For the project this isn’t a problem since the timestamps will be generated on the API side.

# Motion- and smoke sensor

For security reasons we will also be using a motion- and smoke sensor. The sensors have to be powered using a 12Volt power supply. The sensors are also equipped with an alarm relay. When an alarm is triggered the Raspberry Pi will capture this alarm. We achieve this by connecting a cable from the 3.3V pin on the pi to one side of the relay, the other side of the relay will be connected to a GPIO pin. It is possible with the sensors to put the alarm relay in NO (normally open) or NC (normally closed) state. We chose for a normally open state. This way when an alarm is triggered by the sensor the relay will close and the signal from the Pi will be captured on the GPIO pin.

The sensors have the following datasheets:

[Smoke sensor](https://produktinfo.conrad.com/datenblaetter/750000-774999/751559-an-01-ml-DECKENBEWEGUNGSMELDER_360GRA_de_en_fr_nl.pdf)

[Motion sensor](https://produktinfo.conrad.com/datenblaetter/750000-774999/751118-an-01-ml-12V_OPT_RAUCHMELDER_de_en_fr_nl_it.pdf)



# Transport protocol

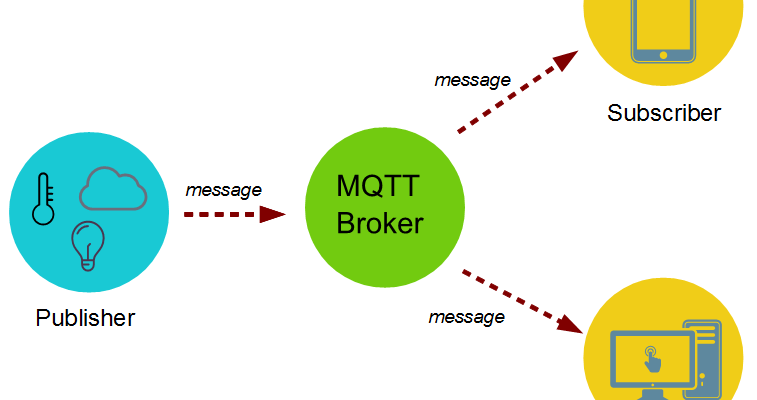
To transmit our data from our Raspberry pi to the application we need to choose a messaging protocol that fits best within our project. In this chapter we compare a couple of these protocols that are out there today.

## MQTT vs AMQP

### MQTT

MQTT stands for Message Queue Telemetry Transport and is an open standard messaging protocol. Its design principles are to minimize network bandwidth and device resources requirements. Two requirements that are necessary regarding IoT solutions. The protocol is designed to be lightweight enough that it can be supported by some of the smallest measuring and monitoring devices, and it can transmit data over far reaching networks.

MQTT is a publish/subscribe protocol that uses a client/server model. Every sensor functions as a client who publishes his data on a specific channel on the server. Other clients can subscribe to this channel and receive the published data. It is possible as a client to subscribe to multiple channels.



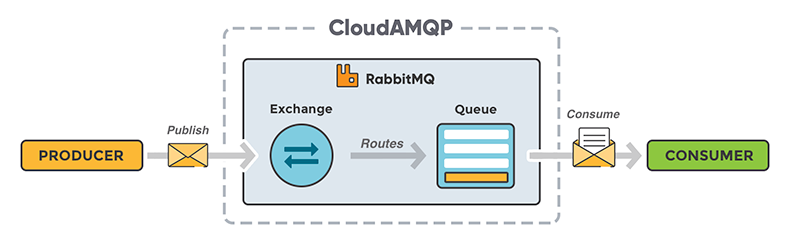
To ensure a secure connection TLS can be implemented. TLS stands for Transport Layer Security and is based on certificates. It allows the connection between de client and the server to be encrypted. By encrypting the communication, we ensure that no third-party can read or tamper with the data that is being exchanged on our connection with the server.



Apart from TLS is also possible to encrypt the message payload using encryption keys, both publishing and subscribing client need to have the same key to decrypt the message. It is also possible to add username and password verification when subscribing to a channel.

### AMQP

AMQP stands for Advanced Message Queuing Protocol and is also an open standard application layer protocol. AMQP enables encrypted and interoperable messaging between applications and is based on a client/server messaging model. AMQP is efficient, multichannel and secure. AMQP has more features related to messaging/routing: you can restrict access to queues, manage these queues and more. An exchange decides which queues to place your message on. These messages are routed from the exchange to the queue depending on exchange types and keys.



The protocol offers authentication and encryption by way of SASL or TLS making it a very reliable and secure protocol. All these features make AMQP a good choice for building large scale, reliable and resilient messaging infrastructures.

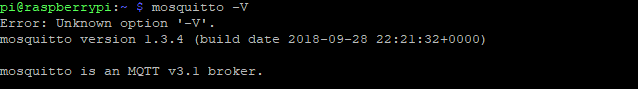
For the project that we will be implementing in the datacenter the MQTT protocol will suit better given it has all the features we need plus it has some big advantages over AMQP: it is very lightweight and consumes very little bandwidth.

## IMPLEMENTING MQTT

First, we need to install MQTT on the Raspberry pi. The program that we need is called Mosquitto. Mosquitto is an open source message broker that implements MQTT and that is suitable for low power single board computers.



This command will install both the Mosquitto service as a broker and as a client. After the installation is complete u can use the next command to see whether the service is correctly installed.



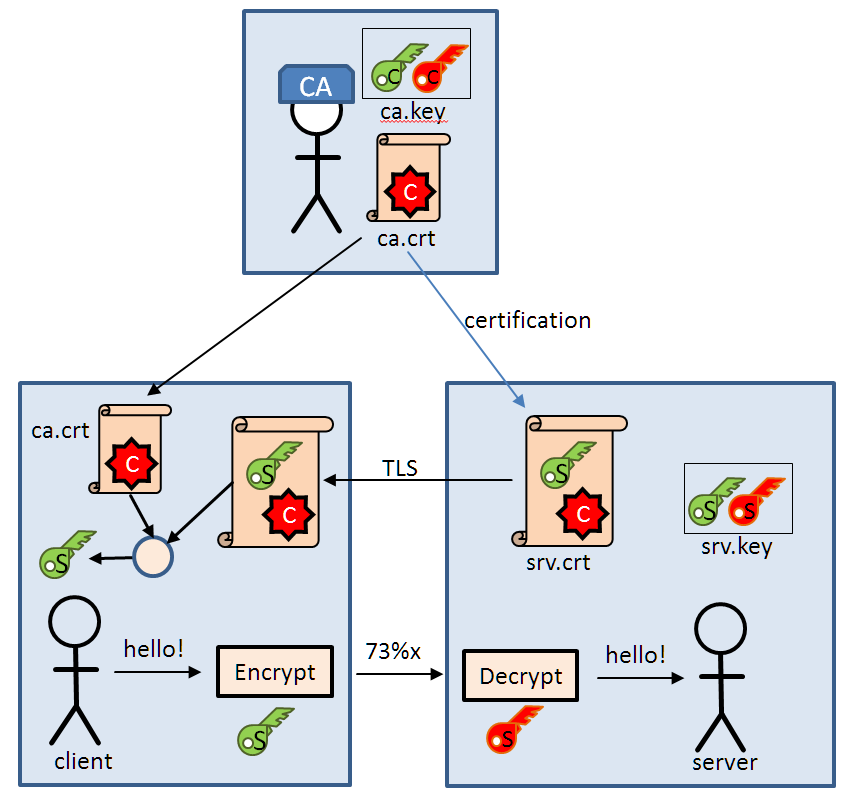
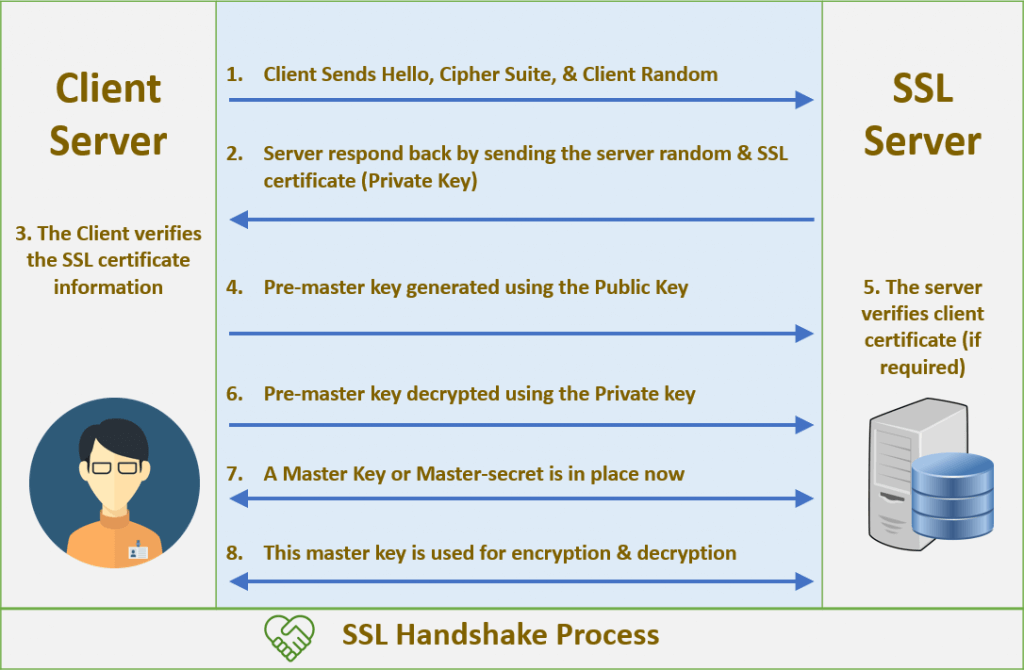
Now we can use the Raspberry pi as a broker/server. We can publish messages to the broker from a client and another client can subscribe to these messages. In our project the client that publishes the messages will be the Raspberry Pi and he will also function as the broker.

After installing the Mosquitto service we need to install Paho. Paho is an MQTT Python client library which we will use in the client programs to publish/subscribe.



## TLS

By default, MQTT doesn’t use encryption but it is possible to secure to connection using TLS. In the next chapter we will explain how TLS should be implemented on both server and client side. TLS works with a client/server authentication and message encryption. First, we will the TLS handshake:

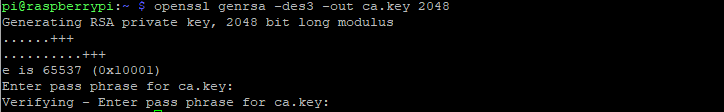


1. The client sends a “hello” message to the server because it wants to start a secure transaction. Next to the “hello” it also sends its cipher suits, a set of cryptographic algorithms, and his compatible TLS version.
2. The server responds and sends it certificate to ensure the client that he is a secure server. The public key is included within the certificate.
3. The client first verifies the certificate, then encrypts a pre-master (shared secret) key using the public key it receives from the server. The server can decrypt this pre-master key using his private key. The pre-master key can only be decrypted with a private key that is stored on the server so no one else but the server can decrypt the data. The server always keeps the private key to himself. This is called asymmetric encryption. After this is done the newly made master key will be used to encrypt and decrypt the information.
4. Both parties now know who they are talking to. Once the verification is over, the encryption takes place to the master-key only, this is symmetric encryption.

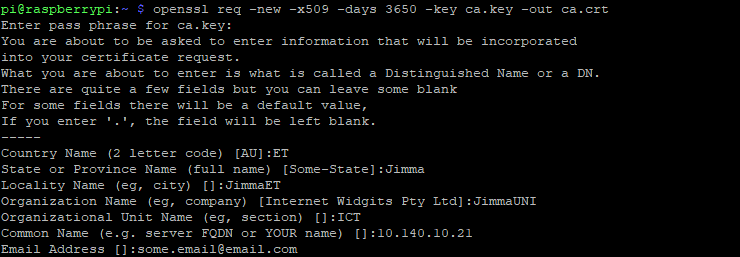
## IMPLEMENTING TLS

### Certificate Authority

First, we need to generate a key pair for the certificate authority (CA). If you want, you can secure this key with a password (raspberry).



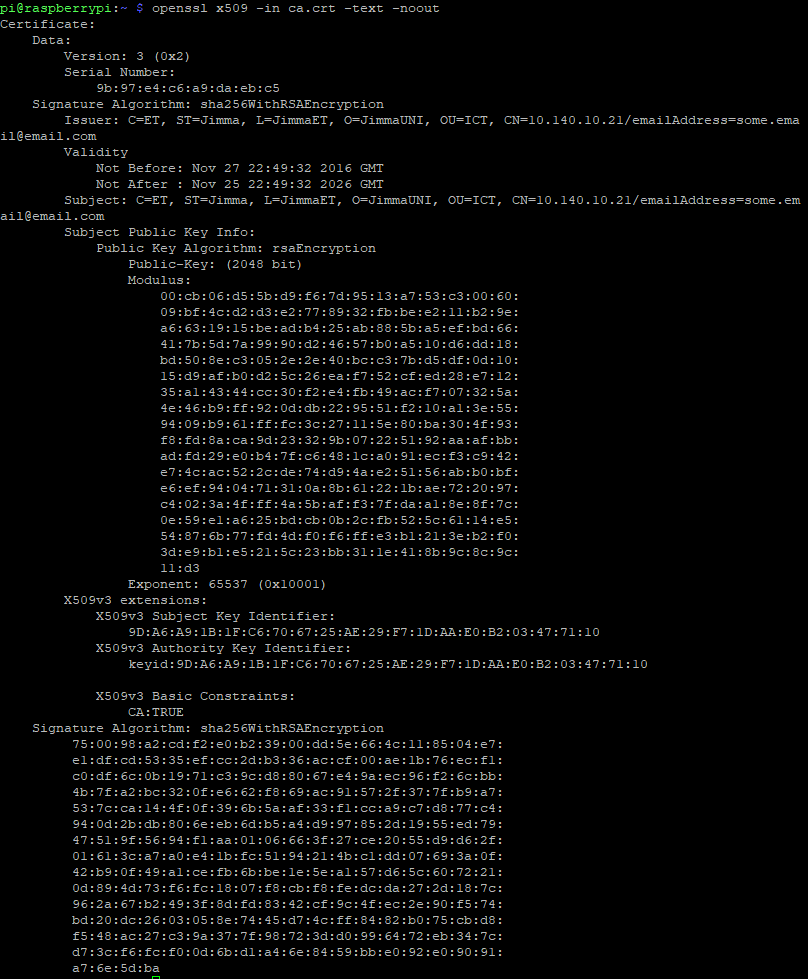
Next, we create a certificate for the CA using the CA key we created. The -x509 stands for a self-signed certificate. We use this used to generate a self-signed root CA. with this certificate we can sign other certificates later. The days we specify are the days that the created certificate will be valid. After you entered the command you will be prompt for certain information. Most of these fields are just for information purposes. An important field is the Common name field. Here we fill in the ip-address of the Pi since this will be the device we will be contacting later as server.



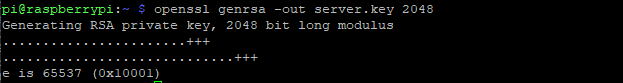
When you open the certificate, you see the following:



You can view information about this certificate with the following command:

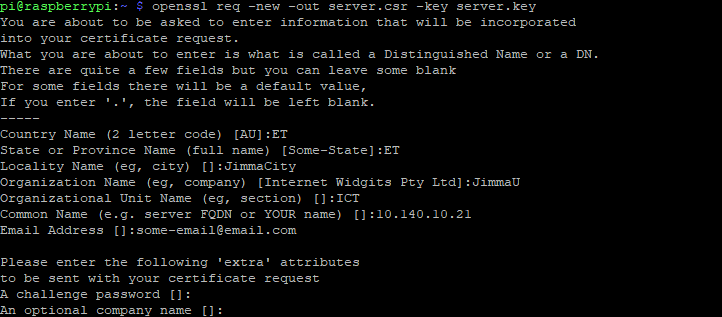


After this we create a server key pair that will be used by the broker (mosquitto server).

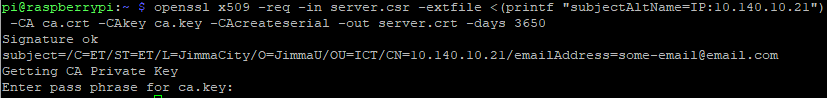


### Server Certificate

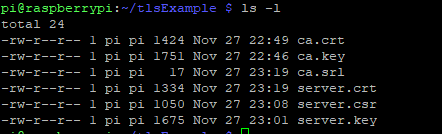
Now we create a certificate request for the server. Again, when filling in the form u should use the ip-address of the pi as Common Name since this will be our server. Also, it’s important to not completely use the same information as you did before otherwise it will cause problems. SSL/TLS will think the CA and Server are the same and will compare ca.crt and server.crt. Since they have different thumbprints the connection will fail. In addition, also don’t use a password to protect the key because the broker won’t be able to decode it.



Next, we use the CA key to verify and sign the server certificate. Since we use an ip-address as common name and not a domain name we need to add this to the command with the -extfile option.



After these steps the following files are created:



When we installed Mosquitto it created a directory on the pi where the configuration is stored and where you can add config-files or certificates.

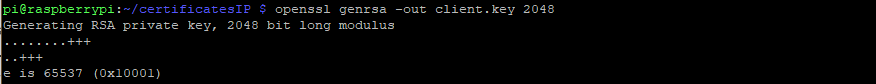


Copy the ca.crt file to the ca\_certificates folder and the server.crt- and server.key files to the certs folder. Also make sure that you copy the ca.crt file to the client. He will need this are will not be able to connect to the server!

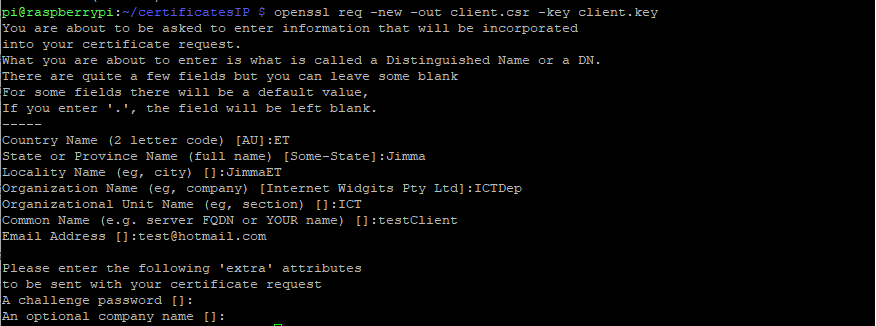
### Client Certificate

We can add extra authentication by using a client certificate. Here is how to create this:

First, we create a client private key:

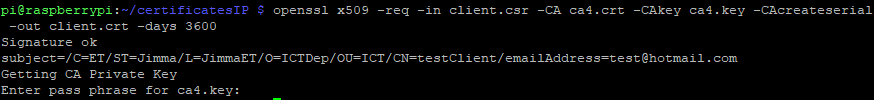


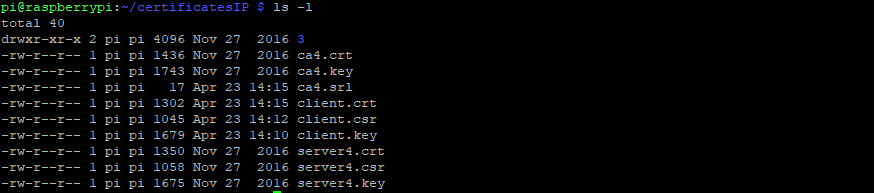
Next, we create a certificate request and we use the client private key to sign it. The most important field is the common name field. Here we enter the name of the client so that the server knows who he is talking to.



Afterword’s we complete the request and the create a client certificate.

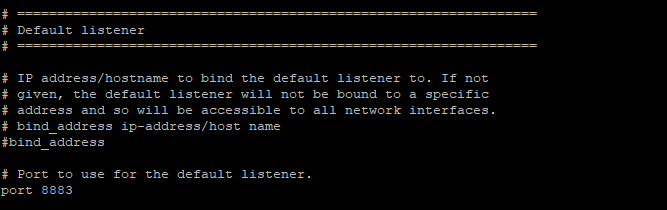
openssl x509 -req -in client.csr -extfile <(printf "subjectAltName=IP:10.140.4.160") -CA ca4.crt -CAkey ca4.key -CAcreateserial -out client.crt -days 3650



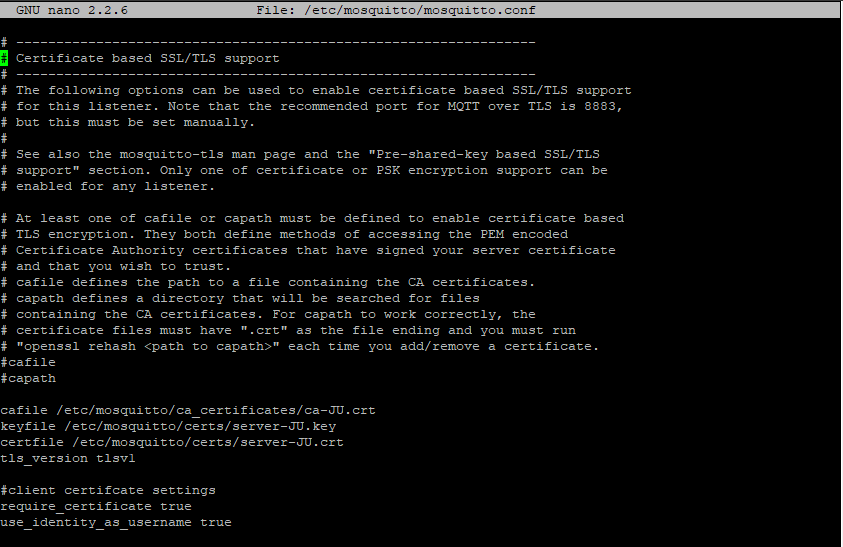
Now the directory will look like this: 

### Mosquitto.conf file

Now edit the configuration file mosquitto.conf. It is also possible to add additional configuration files in the conf.d folder. Make sure you give these files the .conf extension. If you do a first installation it is possible that the mosquitto.conf file is almost empty and just includes a path to the conf.d folder. You can find the complete file online. For our project we only must make a few minor adjustments. In the section default listener: change port 1883 to 8883. 1883 is default port used by MQTT, 8883 is used when secured with TLS.

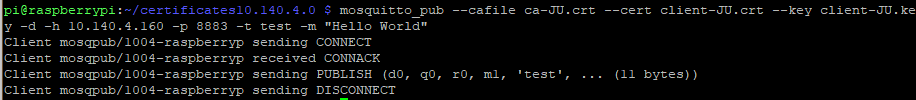


Next, in the section Certificate based SSL/TLS support add the paths to the certificate files u created earlier. Also add which version of TLS u use and tell the Mosquitto broker only to accept clients that have a valid client certificate.

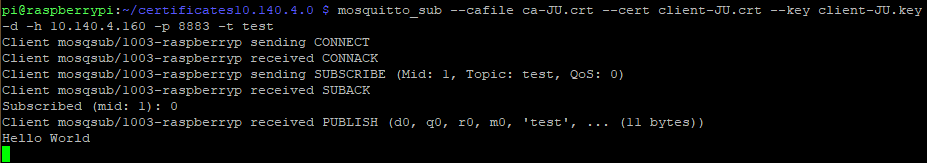


## Example on the command line:

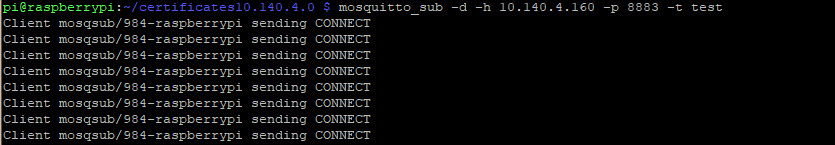
Publish



Subscribe:

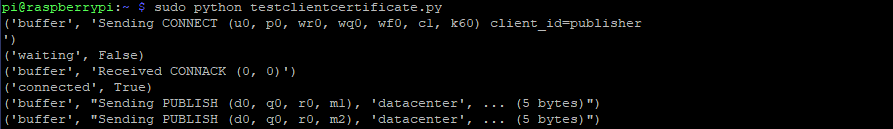


If you don’t add the paths to where you can find the client certificates the server will not acknowledge to connection with the client:

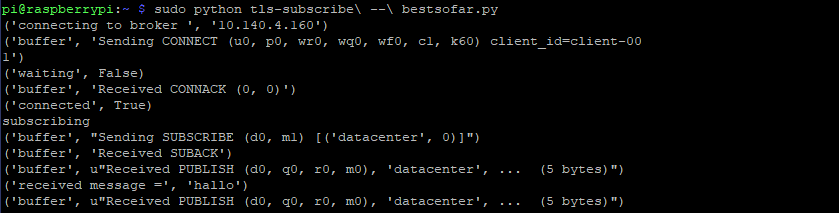


## Example from program code:

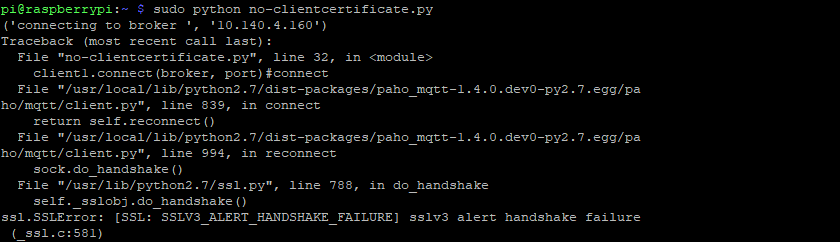
Publish:



Subscribe:



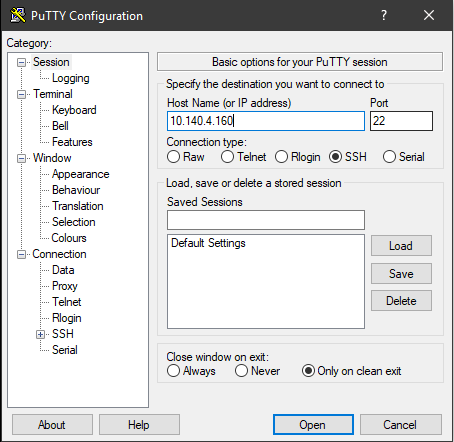
If you do not add the paths in your code, there will be an error saying that there is a handshake failure.



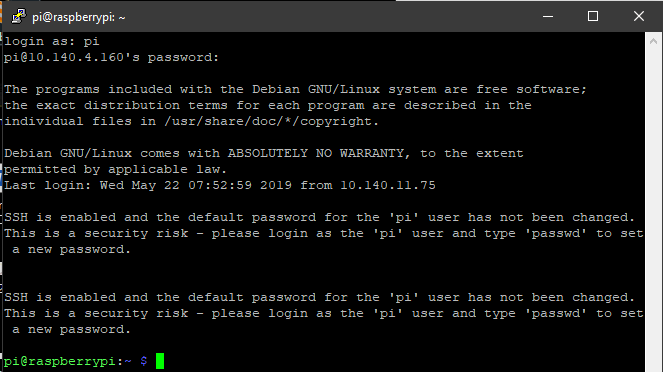
# RASPBERRY PI

## Logging in

To login to the raspberry pi you will need a terminal program (f.e. putty). In the configuration screen you enter the ip address of the Pi: 10.140.4.160 and select the SSH option.



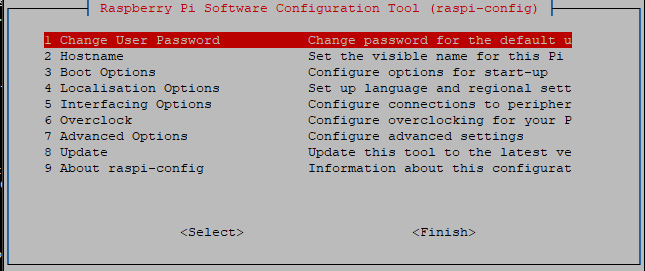
After you click “Open” a login screen should appear. Type here the default login credentials of the Pi: username pi, password raspberry.



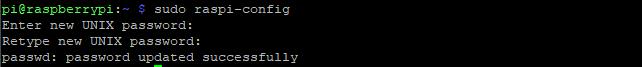
For security reasons it’s bet you change the default password. To change the user password type:



Then choose the first option:



You will be prompt to enter a new password



## The Code

First, we import some libraries which we will need to perform certain functions:

Serial is used to read out the data from the I²C master that is transmitted via a serial cable. Time is used to build in functions regarding time, f.e. wait times. Paho is the MQTT client which we have explained earlier. The GPIO library is to get/send data from/to GPIO pins on the Raspberry pi. And finally, we import the easy SNMP library to perform queries on the PDU’s.

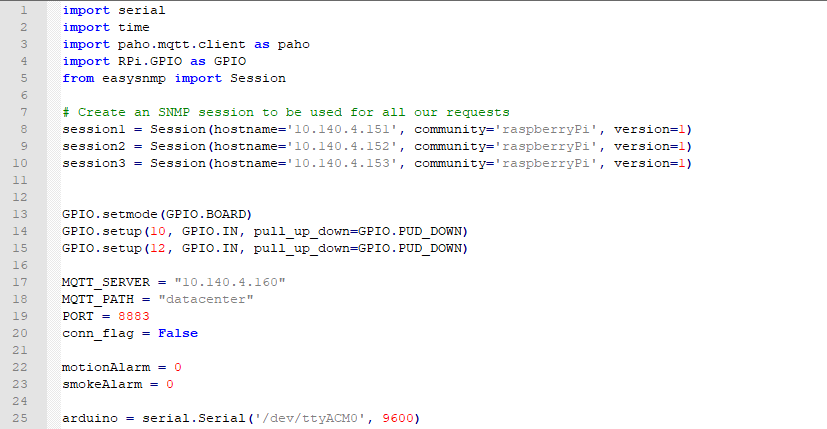
We begin by creating sessions for each of the PDU’s. A session consists of a hostname which is the ip-address of the PDU, a community name: this is the name we configured in the PDU which specifies the account that has the right to read information from the PDU and lastly the version of SNMP.

Then we setup GPIO. We will use the board numbers of the pi to collect data. The 2 pins that we instantiate will receive the info from the smoke- and motion sensor. We use the pull\_up\_down parameter to assign a defined value to the pins until it gets overridden by a stronger force.

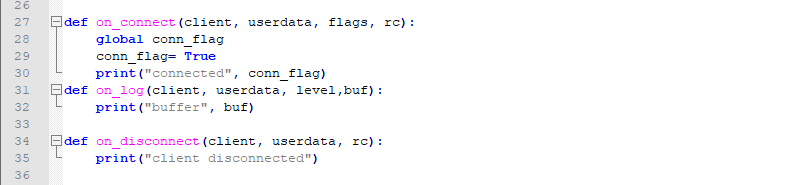
Afterword’s we assign some values to the variables regarding the MQTT configuration. Our server/broker gets the ip-address of the Raspberry pi. The path stands for the channel to which we will publish our data. The port is the secured MQTT port 8883. Lastly, we set a flag that we will use later when trying to connect to the broker.

Next, we assign a value to the 2 alarm variables. This value will be adjusted when an alarm is picked up by one of the sensors.

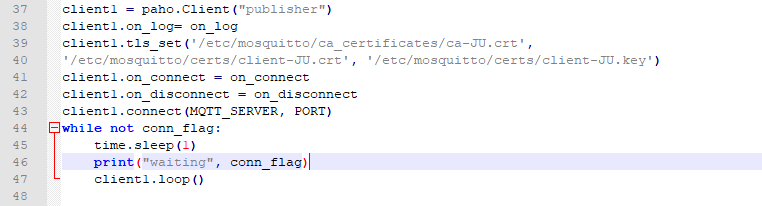
Finally, we instantiate a variable called Arduino. This variable receives the info from the Serial line. The first parameter is the path to the location where the Serial cable receives the Serial data, the second parameter is the baud rate of the Serial line. Make sure that this is the same baud rate as that from the Arduino!



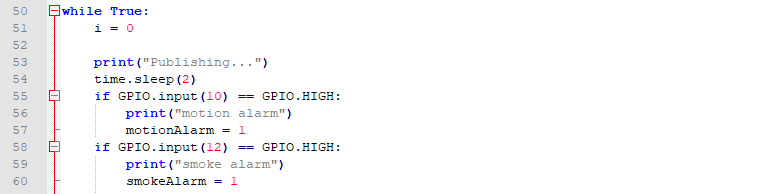
Next, we create a couple of functions that are useful to check connection status of the device and info regarding the data that is been transmitted.



Then we create a client using the Paho library and we try to connect with this client to the broker. We include TLS to the connection be adding the path to the certificates. In the connect function we add the parameters regarding the name of the server and the port. In the while loop we keep trying to connect to the broker if by some reason this would not work the first time.



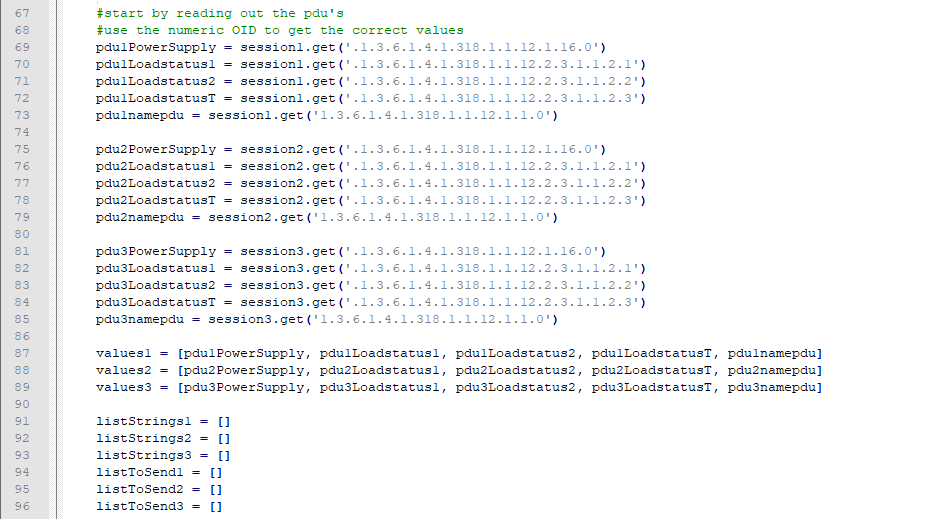
After we’re connected we enter the main program. First we check if theirs an alarm generated by the smoke- or motion detector.



The next step is to read out the correct OID’s from the different PDU’s. The OID’s can be found in the MIB of the PDU of your choice. We are working with old PDU’s, therefore there is only a limited amount of data which can be read from the PDU. The values are:

* The rPDUIdentDevicePowerWatts. Getting this IOD will return the power in Watts.
* The rPDULoadStatusLoad. Getting this OID will return the phase/bank load measured in tenths of Amps. There are two banks on each PDU therefore we can get the value of each of these and the total value as well.
* The rPDUIdentName. Returns The name of the Rack PDU. This can be useful to determine which PDU is delivering which data when you have more PDU’s installed on one rack.

Once we read out the values we put the them in an array. Each rack gets it’s own array. We also create empty lists wich we will use to deformat the retrieved values of the PDU.



When we don’t deformat the retrieved values the varialbes will look like this:



The only thin we want is the part that is shown behind the value parameter. To this we will need to perform some String functions. Since all of our arrays will have the same length we can use one of the in a while loop to determine the time the while loop needs to run. First we convert the SNMP variable to a string and append them to the correct list. Next we locate the position of the value between ‘ ‘. We use the function find. Here is how it works:

string.find(value, start, end)

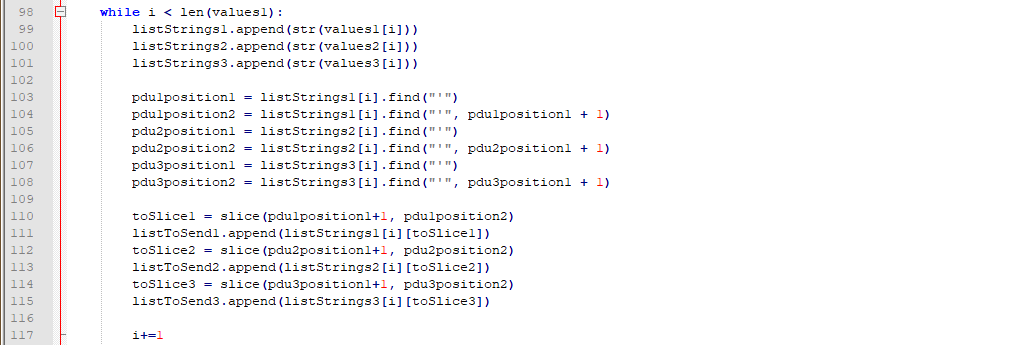
With the find function we search for the first ‘. This will return the index of the symbol. We put this received index in a variable. Since we want the value between ‘ ’ we need to look for the second ‘ as well. We do this in the next line of code. The second value in the find function defines the startpoint from which it should start to search for the symbol. Now we have the two indexes that we need to retreive the value between them. We do this for each pdu because the values will probably not always have the same length.

After this we use the slice function to retrieve the value. Here is how the slice function works:

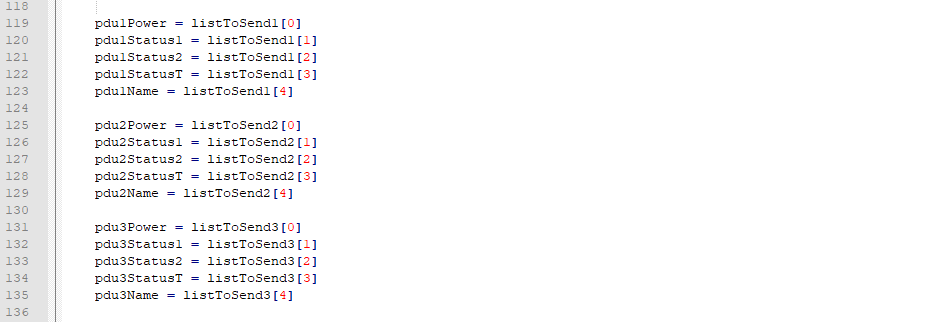
slice(start, end, step)

In the first parameter we put the position of the first symbol +1 because otherwise it will include the symbol to the value. The second parameter is the position of the second symbol. The end parameter will not be included so we don’t need to add +1.

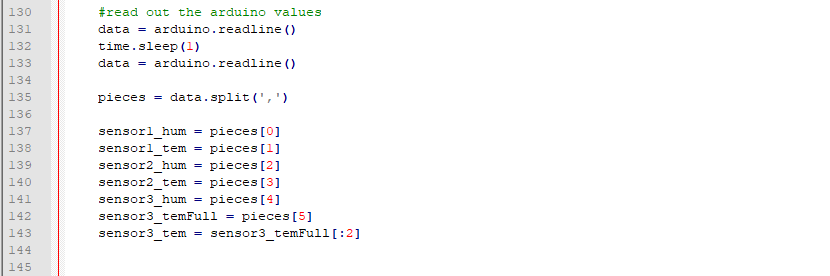
Finally we append only the slice of the complete snmp-string that we want to the list: listToSendi.



Now all we have to do is put the correct values of the list to the rigth variable.



Next we read out the data from the Serial line. I faced problems doing this only once so I did in the way that is showing in the code. The data that is received is deviced by commas so we should split the data on these commas. The data always arrives in numerical order of the slaves from the I²C bus, first humidity then temperature. Knowing this we can put the data in the correct variable. Only for the last value I had to take the first two digits because they were followed by a carriage return: \r\n.



After this we create a string in Json format that can be transmitted, making use of the Json format makes it easier on the API side to put the data in the correct locations in the database.

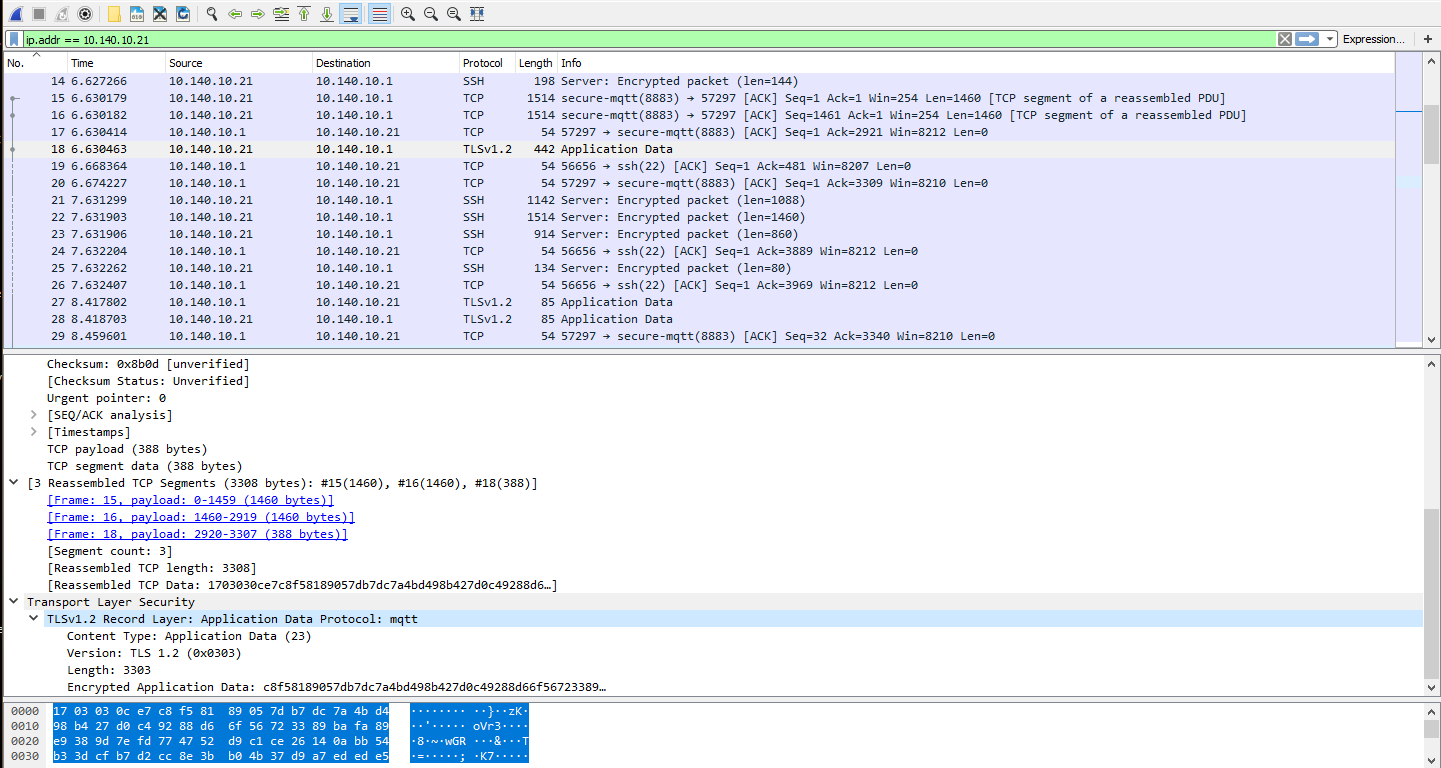


With the publish function we send the data to the server, the parameters we add here are the channel we want to publish the data to and, ofcourse, the data itself. We also put the alarmvalues back to zero.



# testing the connection

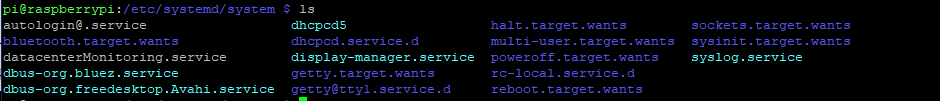
When we check the connection with Wireshark, we can see that the packages are encrypted with TLSv1.2 and that we send the data on the secure-MQTT port 8883.



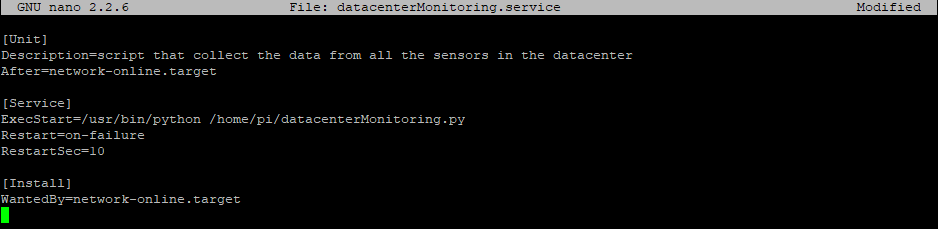
# AUTOSTART SCRIPT AT BOOT

Like a computer the functions of the raspberry pi rely mostly on human input. This is not useful when it comes to the monitoring purposes of the project we build. In our case we want the pi to run the script automatically when it boots up. The best way of starting applications on startup is *systemd.* With systemd, you have the benefit of being able to tell Linux to start certain programs only after certain services have started. As a result, it is a very robust tool for initializing your scripts and applications.

First, we must create a unit file. This is a text file that gives information to system about a service. We will create a unit file that starts our program as a service. Go to /etc/systemd/system.



Here we created the file datacenterMonitoring.service. The file looks like this:

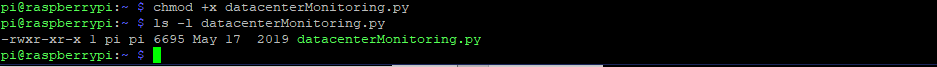


The After keynotes says when our program should run. In our case we will need networking, so we tell the service to start up our program after networking.

Exec Start is the command (or set of commands) used to start our program. Notice that we are using absolute paths to the version of Python we want as well as the location of our program.

In the Install section the Wanted By specifies the target we want our target to be included with.

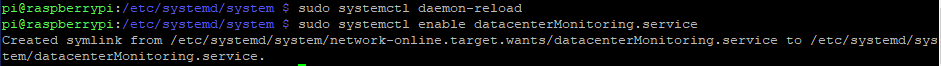
Exit the file and make sure you enable the execution-bits on the script:



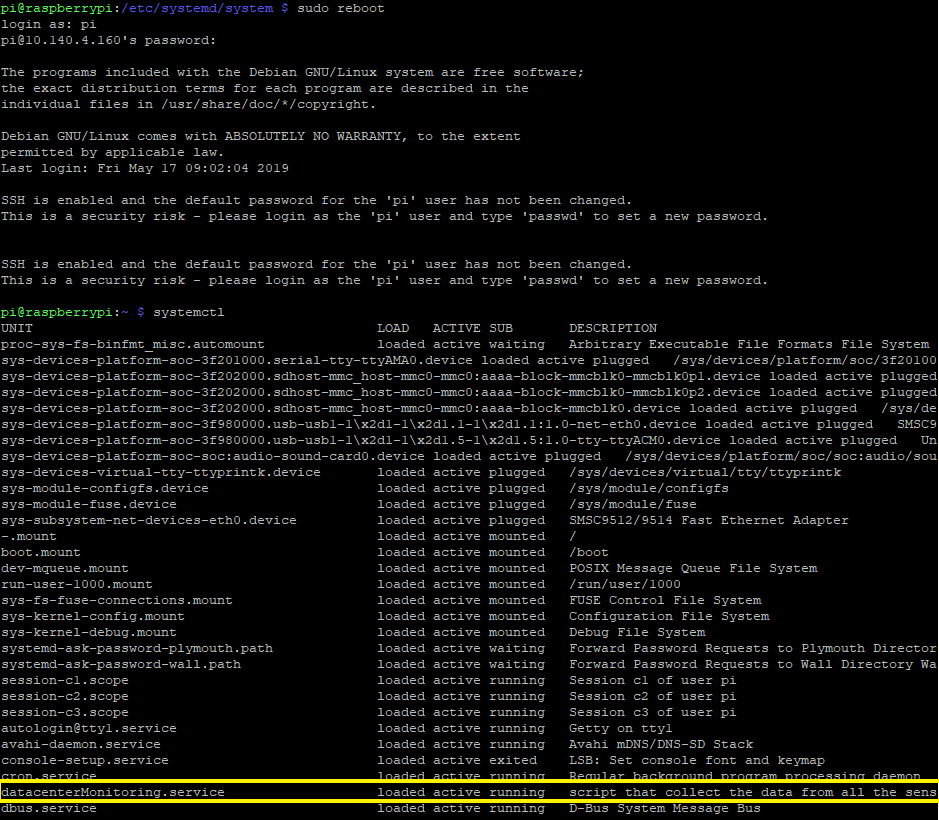
Now enter the following commands:

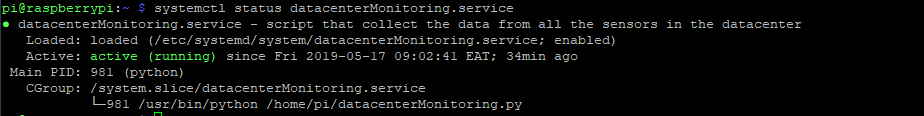
First, we need to tell systemd to recognize our service. You will have to perform the reload command each time you change your service file.

Secondly, we need to tell systemd that we want our service to start on boot.



Now we reboot the pi and verify if the program is running. When you check the services running with systemctl our service should be included in the list.





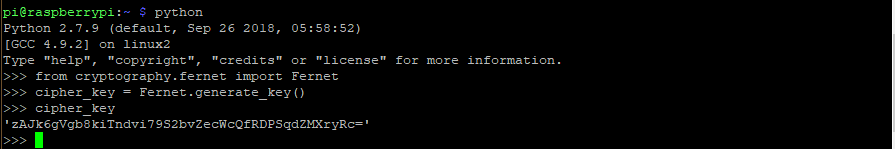
# payload encryption (extra)

Since we are using TLS the data and the connection will be encrypted so there is no need to add this piece of code to our program. The programming that is running does not include payload encryption but if you would like to try it out for future purposes it still can be useful.

First will need to install cryptography



Next, we open a python shell in our Raspberry pi and enter the following commands:



In our main program we add the following library:



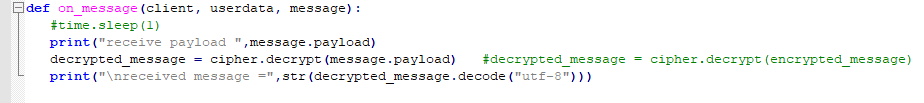
And add the key in the code, do this both on the publisher and the subscriber:



Finally, we will use the generated key to encrypt the message. You must make sure that the message to be encrypted is in bytes. You can do this by putting a ‘b’ in front of the message.

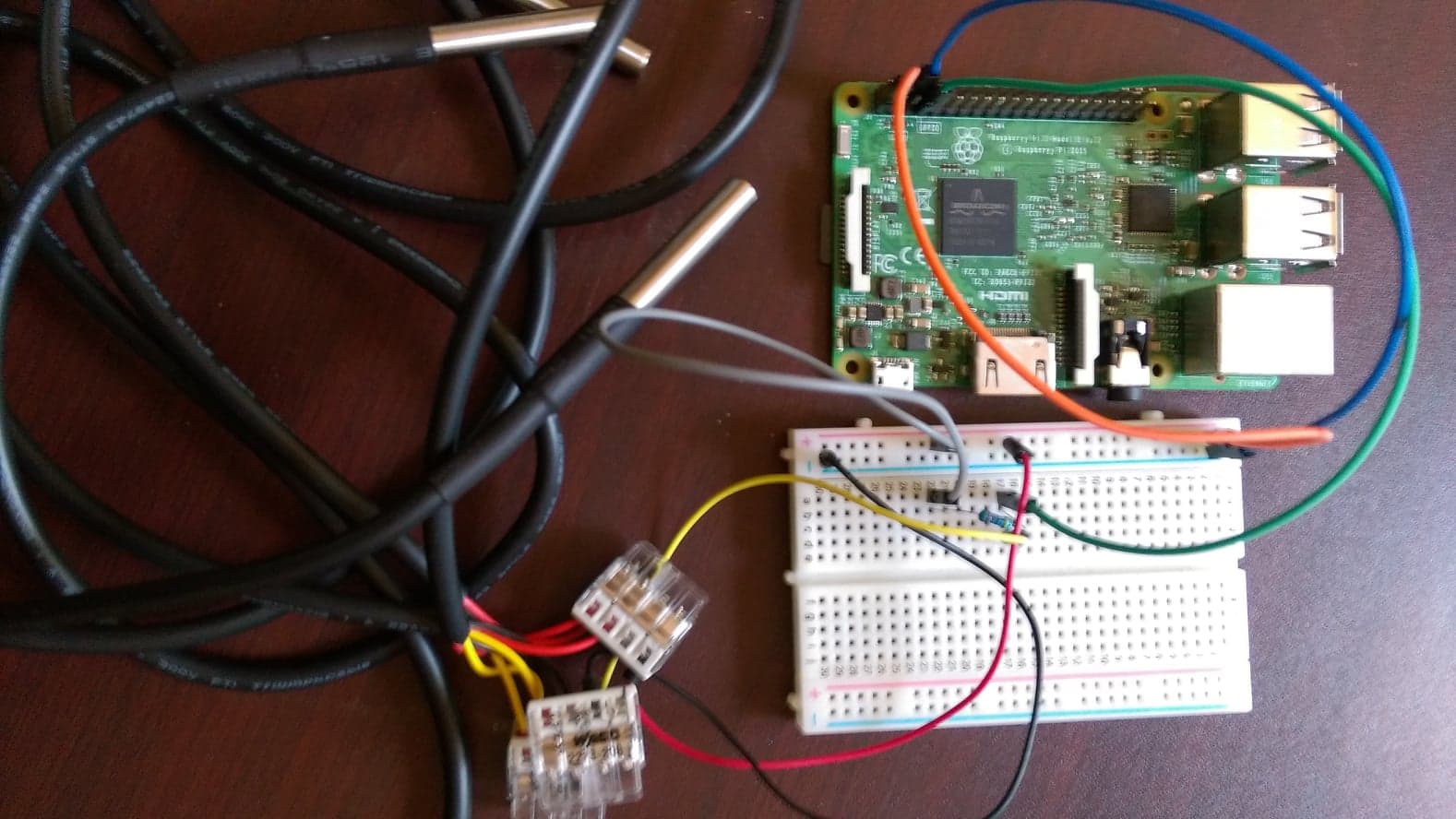


On the subscriber site we can create a function on\_message. This function will decrypt the received message of the publisher:

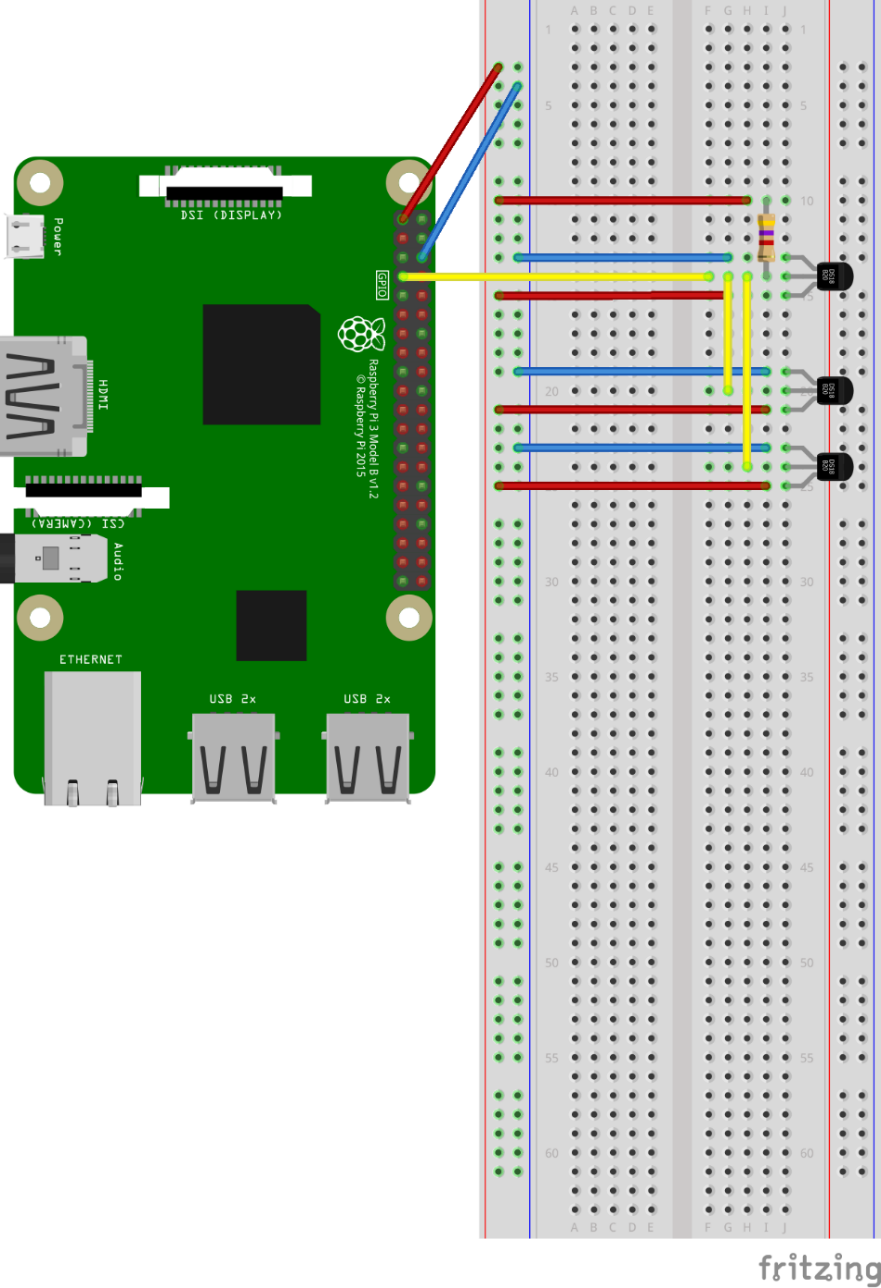


# 1-wire sensors on the Pi (extra)

We also brought different kind of sensors that we will not use in this project. In this chapter I give a short summary on how you can use these sensors on the raspberry pi. They only measure the temperature, for this reason we went with the other sensors. If the university wants to expand their system, they can implement these sensors as well which are more suitable in harsher environments.



## The scheme

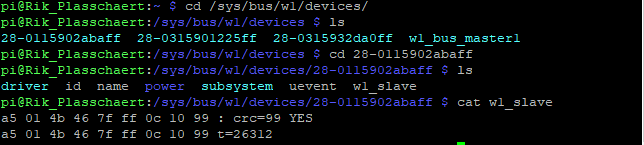


All sensors are connected and use 1 mutual data, vcc and ground pin. The pi delivers the power (3.3v) to the sensors on via the bus. The common data pin is connected to GPIO4. Between de data pins there is a 4.7K pullup resistor. This is used as a safety method to ensure that inputs to the pi are at expected logic levels.

To see which and how many 1 wire sensors are connected to the Pi go type:

The device names start with 28- ….

As shown in the pictures 3 devices are connected to the 1 wire interface of the Raspberry Pi, if we connect more sensors via the same data line these sensors will also be visible here. It is possible to read out one sensor:



1. Go to the correct folder
2. Ask the content of the chosen sensor by typing cat w1\_slave
3. First you see the mac-address of the sensor followed by a crc-check. When it says yes, the sensor is working correct. The second line also shows the mac-address followed by the temperature 🡪 26.312 °C

## The code:

*import os*

*import glob*

*import time*

*class DS18B20:*

*def \_\_init\_\_(self):*

*# load required kernel modules*

*os.system('modprobe w1-gpio')*

*os.system('modprobe w1-therm')*

*# Find file names for the sensor(s)*

*base\_dir = '/sys/bus/w1/devices/'*

*device\_folder = glob.glob(base\_dir + '28\*')*

*self.\_num\_devices = len(device\_folder)*

*self.\_device\_file = list()*

*i = 0*

*while i < self.\_num\_devices:*

*self.\_device\_file.append(device\_folder[i] + '/w1\_slave')*

*i += 1*

*def \_read\_temp(self,index):*

*# Issue one read to one sensor*

*# you should not call this directly*

*f = open(self.\_device\_file[index],'r')*

*lines = f.readlines()*

*f.close()*

*return lines*

*def tempC(self,index = 0):*

*# call this to get the temperature in degrees C*

*# detected by a sensor*

*lines = self.\_read\_temp(index)*

*retries = 5*

*while (lines[0].strip()[-3:] != 'YES') and (retries > 0):*

*# read failed so try again*

*time.sleep(0.1)*

*#print('Read Failed', retries)*

*lines = self.\_read\_temp(index)*

*retries -= 1*

*if retries == 0:*

*return 998*

*equals\_pos = lines[1].find('t=')*

*if equals\_pos != -1:*

*temp = lines[1][equals\_pos + 2:]*

*return float(temp)/1000*

*else:*

*# error*

*return 999*

*def device\_count(self):*

*# call this to see how many sensors have been detected*

*return self.\_num\_devices*

All the functions that are initiated here starting with def can be used in our other script. This will be our main script where we read out how many sensors are available and what their value is.

*# Temp.py*

*from ds18b20 import DS18B20*

*device\_list = []*

*# test temperature sensors*

*x = DS18B20()*

*count=x.device\_count()*

*i = 0*

*#Reading out every sensor from 1wire dir 1time*

*while i < count:*

*#print(x.tempC(i))*

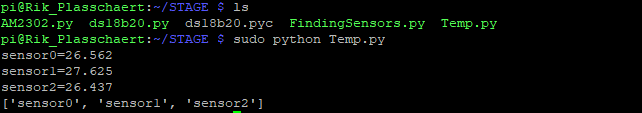
*device\_list.append('sensor'+str(i))*

*print device\_list[i] + "=" + str(x.tempC(i))*

*i += 1*

*print device\_list*

OUTPUT:



Conclusion….

Typ hier je besluit

# Bibliography

Arduino. (2019). *Wire Library*. Retrieved from Arduino: https://www.arduino.cc/en/Reference/Wire

Arora, H. (2015, March 6). *Beginners Guide to Installing, Using, and Configuring Net-SNMP – Part 1*. Retrieved from Maketecheasier: https://www.maketecheasier.com/net-snmp-part-1/

Circitor. (2019, May 1). *PowerNet-MIB*. Retrieved from Circitor: http://www.circitor.fr/Mibs/Html/P/PowerNet-MIB.php

Cohn, R. (n.d.). *A Comparison of AMQP and MQTT*. Retrieved from https://static.dzone.com/dz1/dz-files/StormMQ\_WhitePaper\_-\_A\_Comparison\_of\_AMQP\_and\_MQTT.pdf

Cope, S. (2011-2019). *Steve's Internet guide*. Retrieved from Steve's Internet guide: http://www.steves-internet-guide.com/

Felix. (2018). *Wireless communication between Raspberry Pi’s via MQTT broker/client*. Retrieved from Raspberry Pi Tutorials: https://tutorials-raspberrypi.com/raspberry-pi-mqtt-broker-client-wireless-communication/

Gimian, F. (2015). *Welcome to Easy SNMP*. Retrieved from Easy SNMP: https://easysnmp.readthedocs.io/en/latest/

Gramba, I. (2014, July 17). *Simple I2C protocol for advanced communication between Arduinos*. Retrieved from Berryjam: http://www.berryjam.eu/2014/07/advanced-arduino-i2c-communication/

Heymsfeld, R. (n.d.). *raspberry pi to arduino spi communication*. Retrieved from hobbizine: http://robotics.hobbizine.com/raspiduino.html

Hymel, S. (2016, July 18). *How to Run a Raspberry Pi Program on Startup*. Retrieved from Sparkfun: https://learn.sparkfun.com/tutorials/how-to-run-a-raspberry-pi-program-on-startup/all

Light, R. (2018, September 2). *Paho-mqtt*. Retrieved from Pypi: https://pypi.org/project/paho-mqtt/

Meyers. (2012, Augustus 11). *Raspberry Pi + Arduino + SPI*. Retrieved from mitchtech: http://mitchtech.net/raspberry-pi-arduino-spi/

*Pi Network Monitoring*. (2018, June 24). Retrieved from Fun Tech Projects: https://funprojects.blog/2018/06/24/pi-network-monitoring/

Piper, A. (2013). *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP*. Retrieved from VMWare Blogs: https://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html

Shopper, S. (2008, Januari 13). *The Most Common OpenSSL Commands*. Opgehaald van SSL Shopper: https://www.sslshopper.com/article-most-common-openssl-commands.html

SourceForge. (2015, February 26). *Net-SNMP*. Retrieved from Net-SNMP: http://www.net-snmp.org/

Stoffregen, P. (2019). *Dallas Semiconductor's 1-Wire Protocol*. Opgehaald van Arduino: https://playground.arduino.cc/Learning/OneWire/