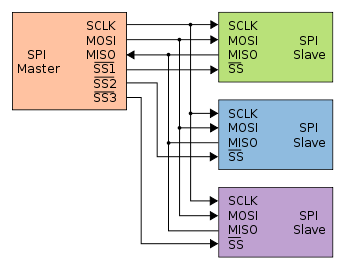
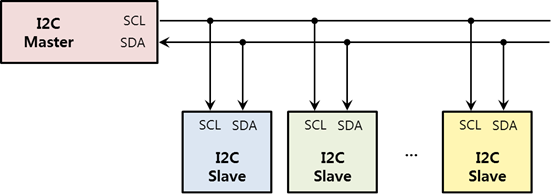
**Configuration for Datacenter monitoring**

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

SPI:



I²C:

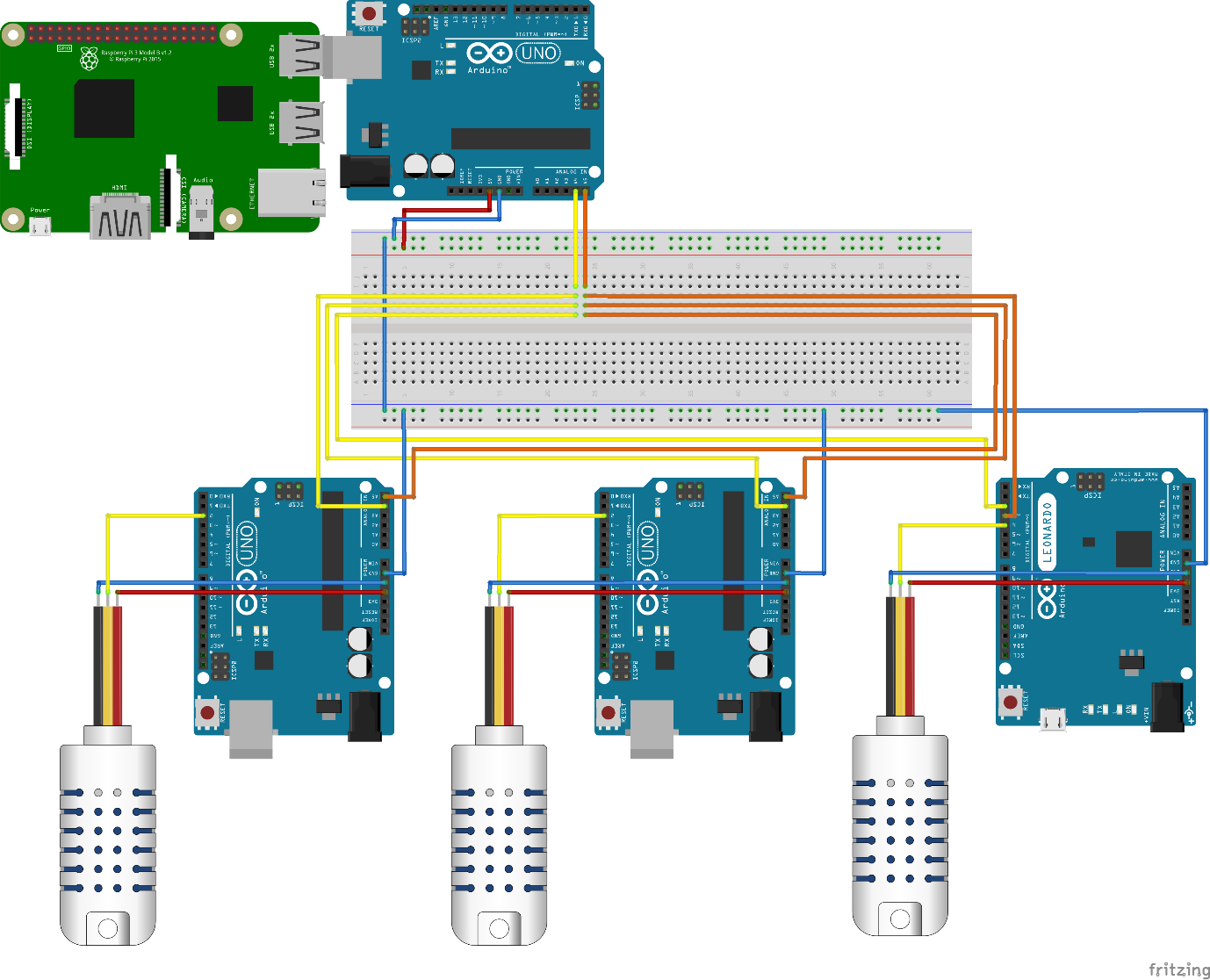


As you can see on the pictures the biggest difference between the two protocols is the wiring. I²C only needs two wires for communicating, one clock- and one data wire. SPI needs four. For each slave that you want to connect to the SPI bus you will need a different SS (Slave Select or Chip Select) line which makes more complex wiring. Whit 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 threw 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 that I²C but it can only do that for short distances while I²C on the other hand can transmit data over much 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 then 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. A part from, using a raspberry Pi in a configuration as this would be overkill since we only want to perform repetitive tasks like control a number of devices and gather data. The Pi has much more resources and consumes a lot more power making it not suitable just for these tasks. Later on we will use the Pi to act as a gateway between ours devices gathering the data and the database where we need to send the data.

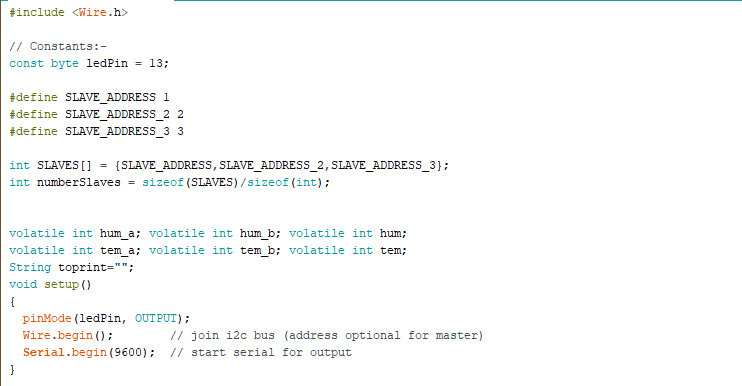
THE SCHEME:



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 (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 send 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.

THE CODE:

MASTER DEVICE

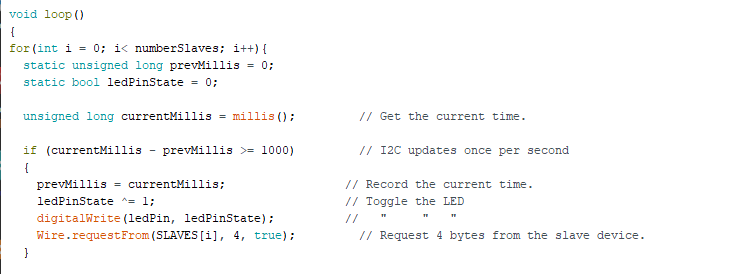


We start our code by including the Wire.h library. This library allows you to communicate with I²C devices. Afterword’s we declare a pin which will toggle when data is being transmitted. Next we declare the addresses of the slave devices connected on the bus. Then we put them in an array which we will 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.

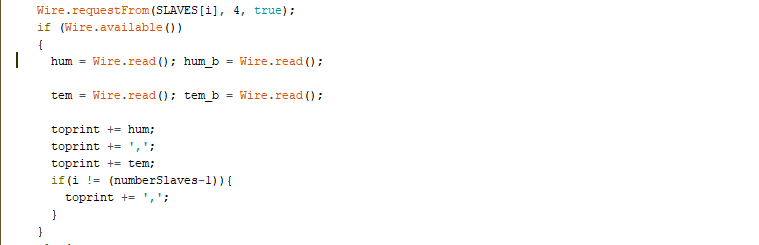
In C code u can get the size of an array in bytes using sizeoff(array), but to get the number of elements you divide this by the size of a single element. That’s why we make use of the division in the part to get the number of slaves.

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 ledpin 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 amount 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 send his data, this is done by putting a Boolean of true as final parameter.

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 represent 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.

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.



You can find the full code on the next pages.

#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);

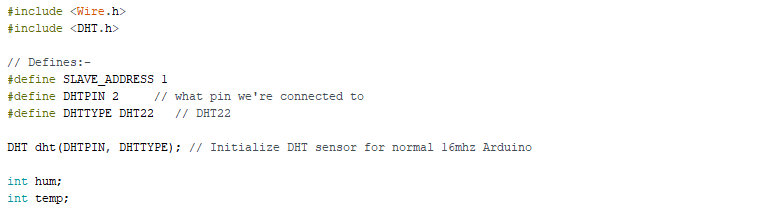
}

Serial.println(toprint);

toprint="";

}

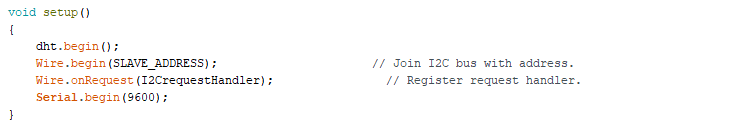
CODE SLAVE:



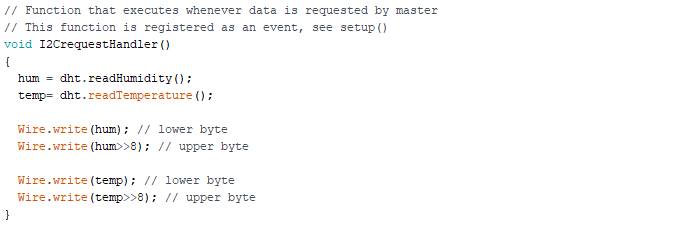
Just like with the master we start our code by including the Wire.h library. This library allows you to communicate with I²C devices. Apart from the Wire.h library we also need to include the DHT.h library. This library gives us the possibility to read out the connected DHT-sensor. It is a digital, one-wire 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 dataline of the I²C bus. Since this data is written in bytes we will need 2 bytes to represent the integer values. E.g. humidiy value = 28: the first byte will represent the 2, the second the 8.

The full code is displayed below:

#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

}

MQTT vs AMQP

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

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 is able to 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 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. The protocol offers authentication and encryption by way of SASL or TLS. Furthermore AMQP has more features related to messaging: you can restrict access to queues, manage these queues and more. These features make AMQP a good choice for building large scale, reliable and resilient messaging infrastructures.

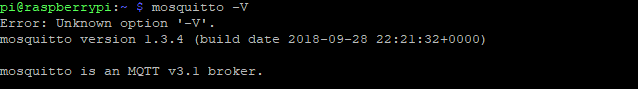
For the project that we will be implementing in the datacenter the MQTT protocol will suit better given it has all the features we need plus it has some big advantages over AMQP: it is very lightweight and consumes very little bandwidth.

IMPLEMENTING MQTT AND TLS

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.

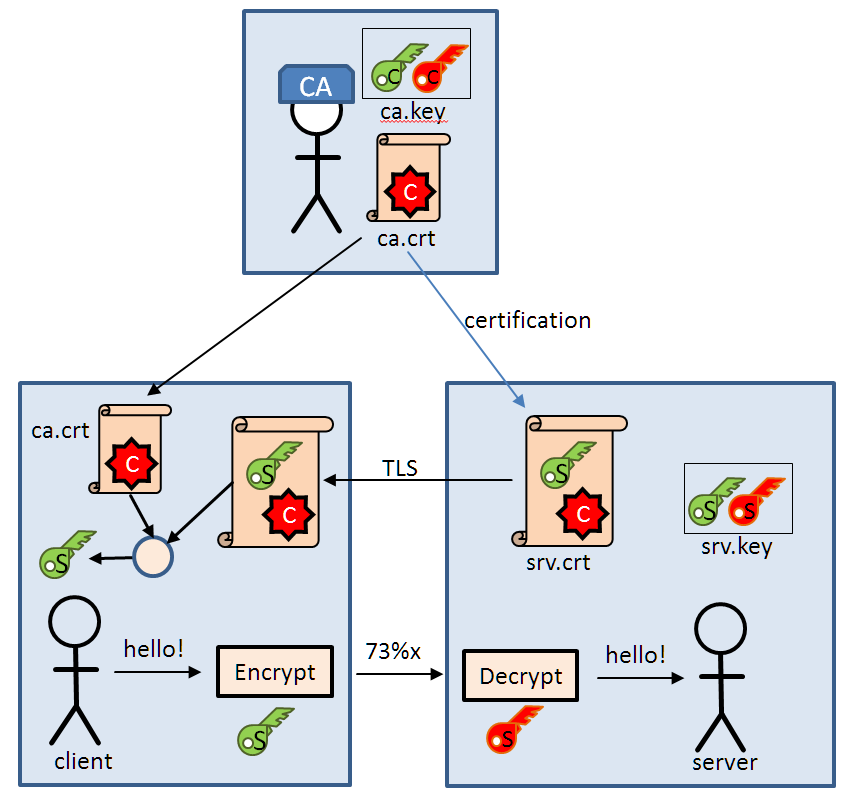
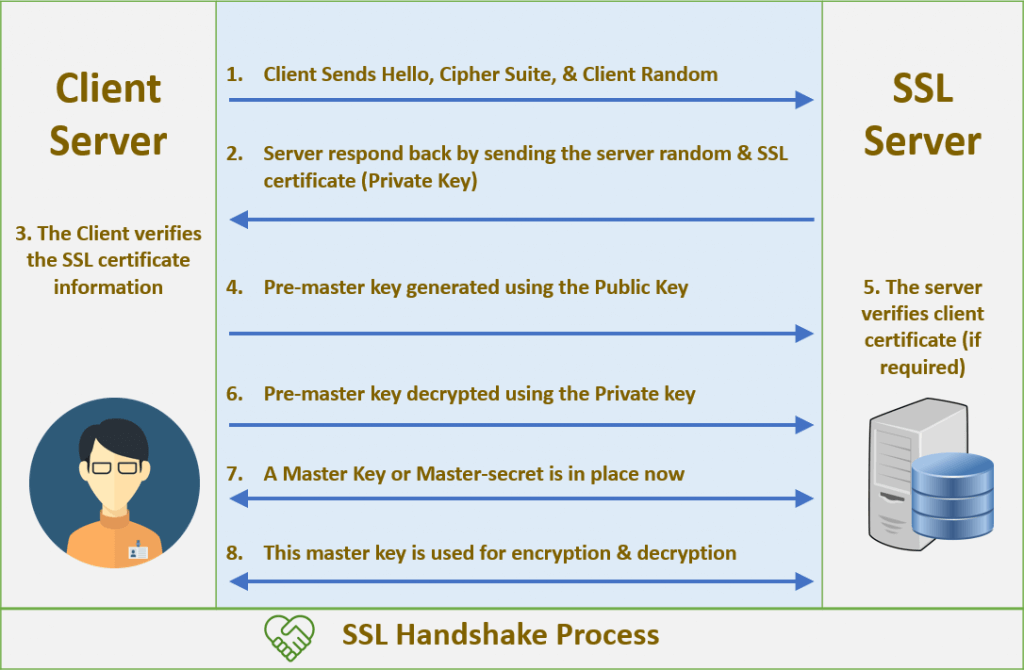


Now we can use the raspberry pi as a broker/server. We can publish messages to the broker from a client and another client can subscribe to these messages. In our project the client that publishes the messages will also be the broker.

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.



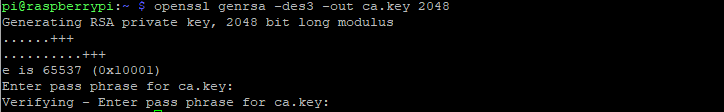
By default MQTT doesn’t use encryption but it is possible to secure to connection using TLS. In the next chapter we will explain how TLS should be implemented on both server and client side. TLS works with a client/server authentication and message encryption. First we will the TLS handshake:



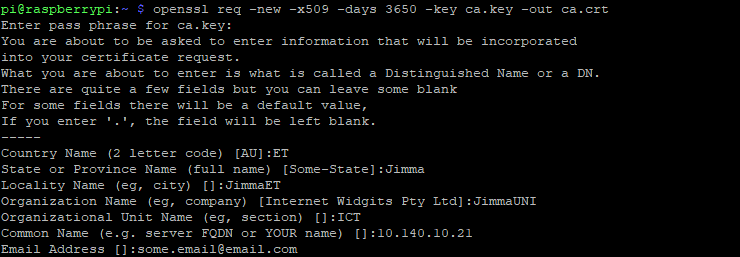
1. The client sends a “hello” message to the server cause it wants to start a secure transaction. Next to the “hello” it also sends its cipher suits, a set of cryptographic algorithms, and his compatible TLS version.
2. The server responds and sends it certificate to ensure the client that he is a secure 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

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



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.

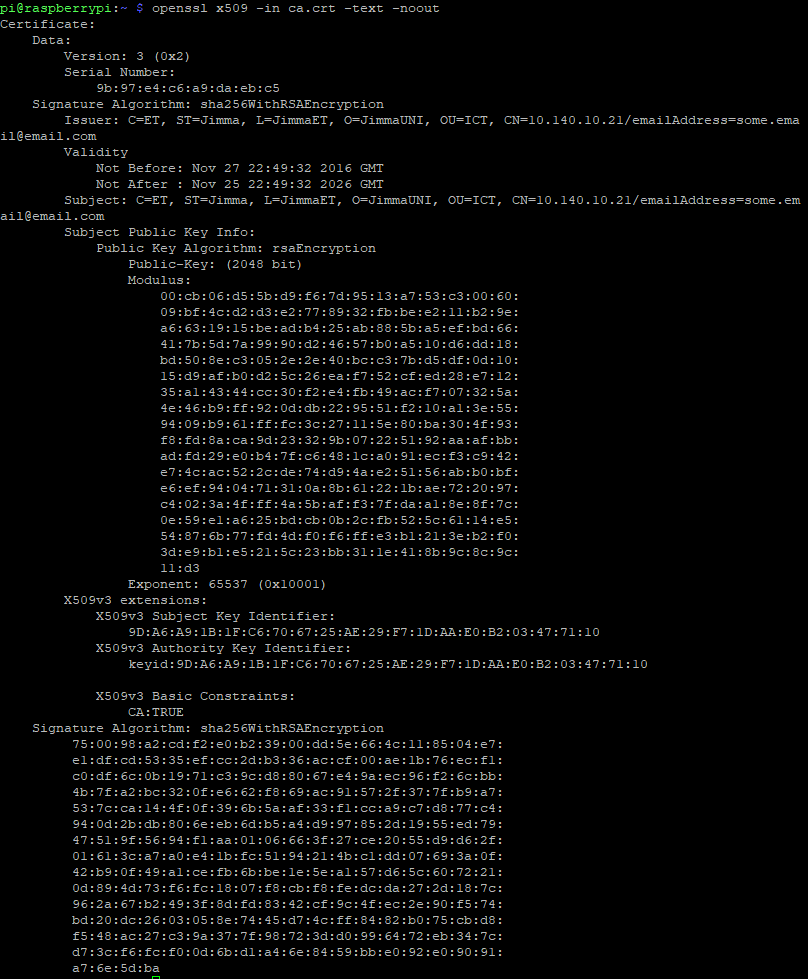


When you open the certificate you see the following:

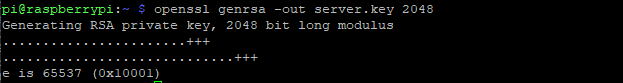


(Of course for security reasons, this will not be the certificate we will be using in our project).

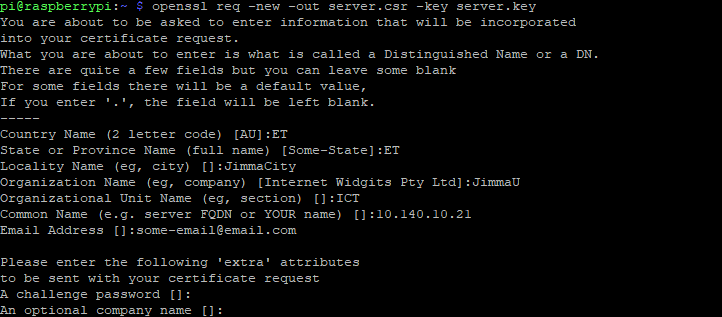
You can view information about this certificate with the following command:



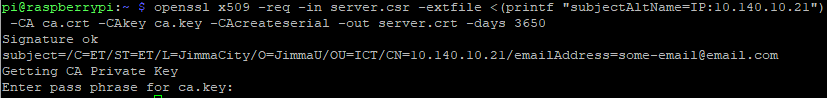
After this we create a server key pair that will be used by the broker (mosquitto server).



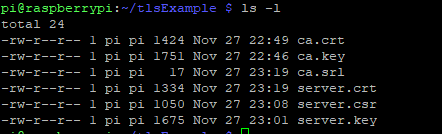
Now we create a certificate request for the server. Again when filling in the form u should use the ip-address of the pi as Common Name since this will be our server. Also it’s important to not completely use the same information as you did before otherwise it will cause problems. SSL/TLS will think 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:



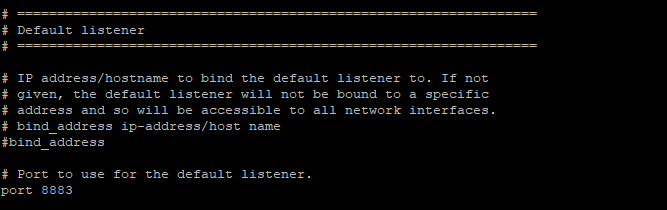
When we installed Mosquitto it created a directory on the pi where the configuration is stored and where you can add config-files or certificates.



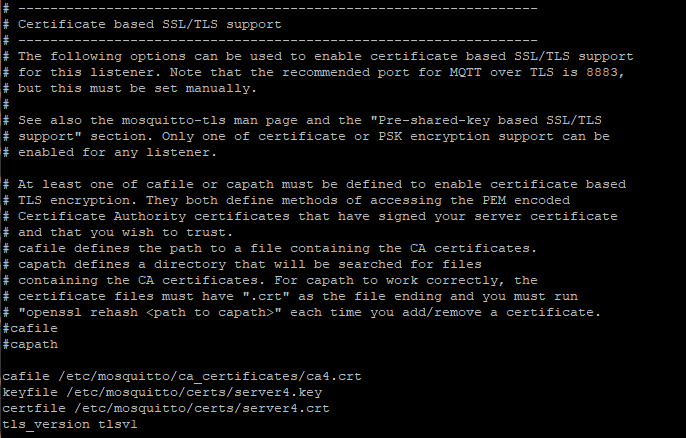
Copy the ca.crt file to the ca\_certificates folder and the server.crt- and server.key files to the certs folder.

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!

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 have to 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. In my case the paths are the ones below:



Also add which version of TLS u use.

Now that we created all the necessary files we can create the python file which will receive the data from the I²C master and sends it via secured MQTT to the broker. This will be explained in the next chapter.

RASPBERRY PI PUBLISHER

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

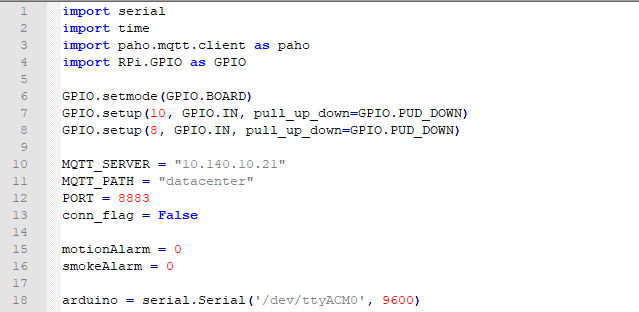
Serial is used to read out the data from the I²C master that is transmitted via a serial cable. Time is used to build in functions regarding time, f.e. wait times. Paho is the mqtt client which we have explained earlier. And finally GPIO to get/send data from/to GPIO pins on the raspberry pi.

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.

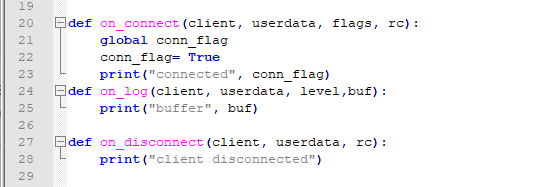
Afterword’s we assign some values to the variables regarding the MQTT configuration. Our server/broker get’s the ip-address of the raspberry pi. The path stands for the channel to which we will publish our data. The port is the secured MQTT port 8883. Lastly we set a flag that we will use later on when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

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

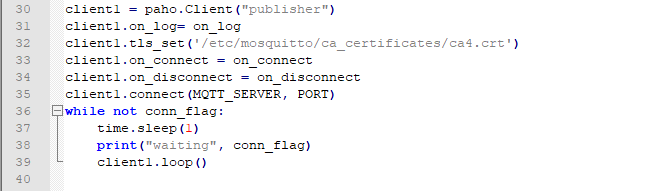
Finally we instantiate a variable called Arduino. This variable receives the info from the Serial line. The first parameter is the path to the location where the Serial cable receives the Serial data, the second parameter is the baud rate of the Serial line. Make sure that this is the same baud rate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

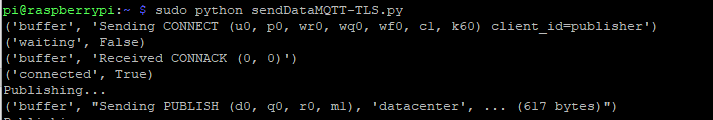


After we’re connected we enter the main program. First will check if theirs an alarm generated by the smoke- or motion detector.

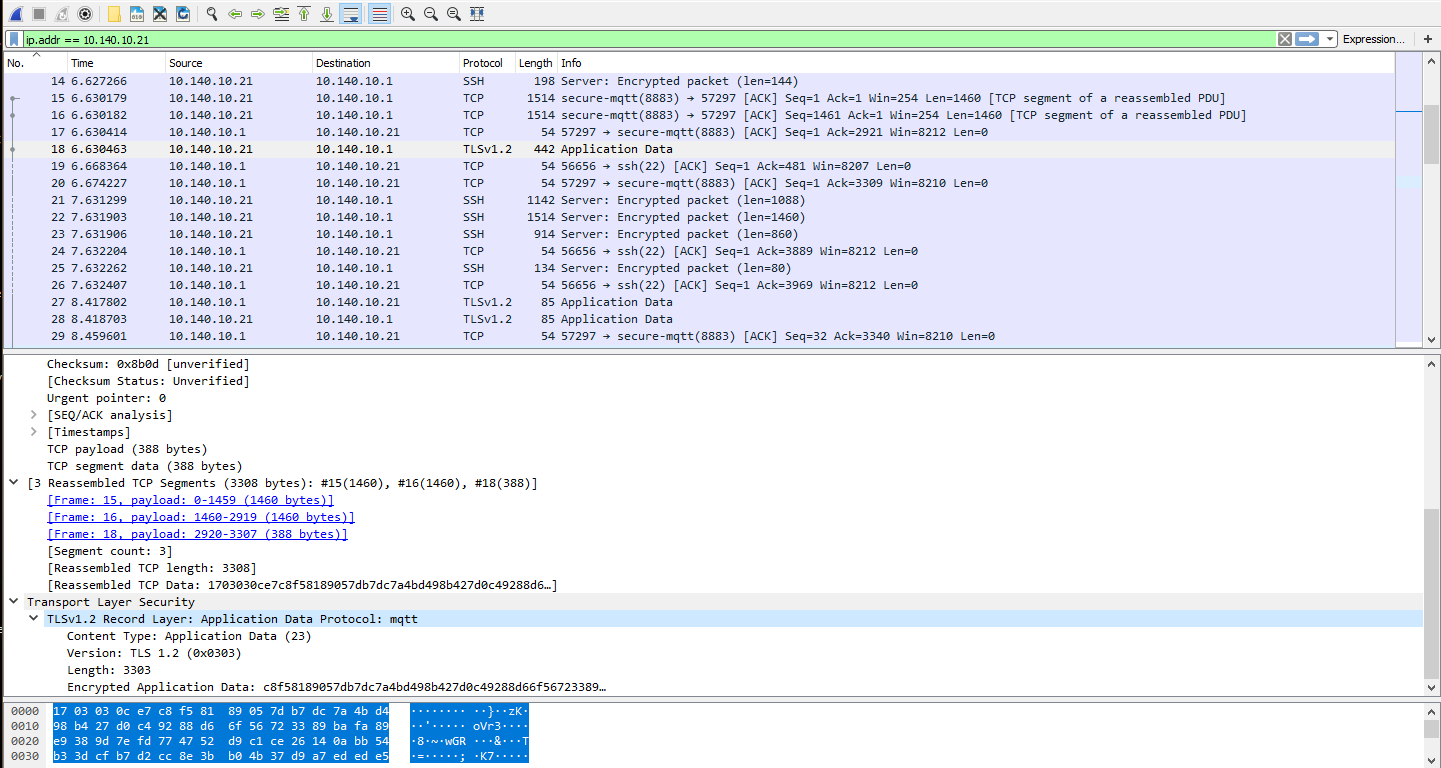
Next we read out the data from the Serial line. I faced problems doing this only once so I did in the way that is showing in the code. The data that is received is deviced by commas so we should split the data on these commas. The data always arrives in numerical order of the slaves from the I²C bus, first humidity then temperature. Knowing this we can put the data in the correct variable.

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.

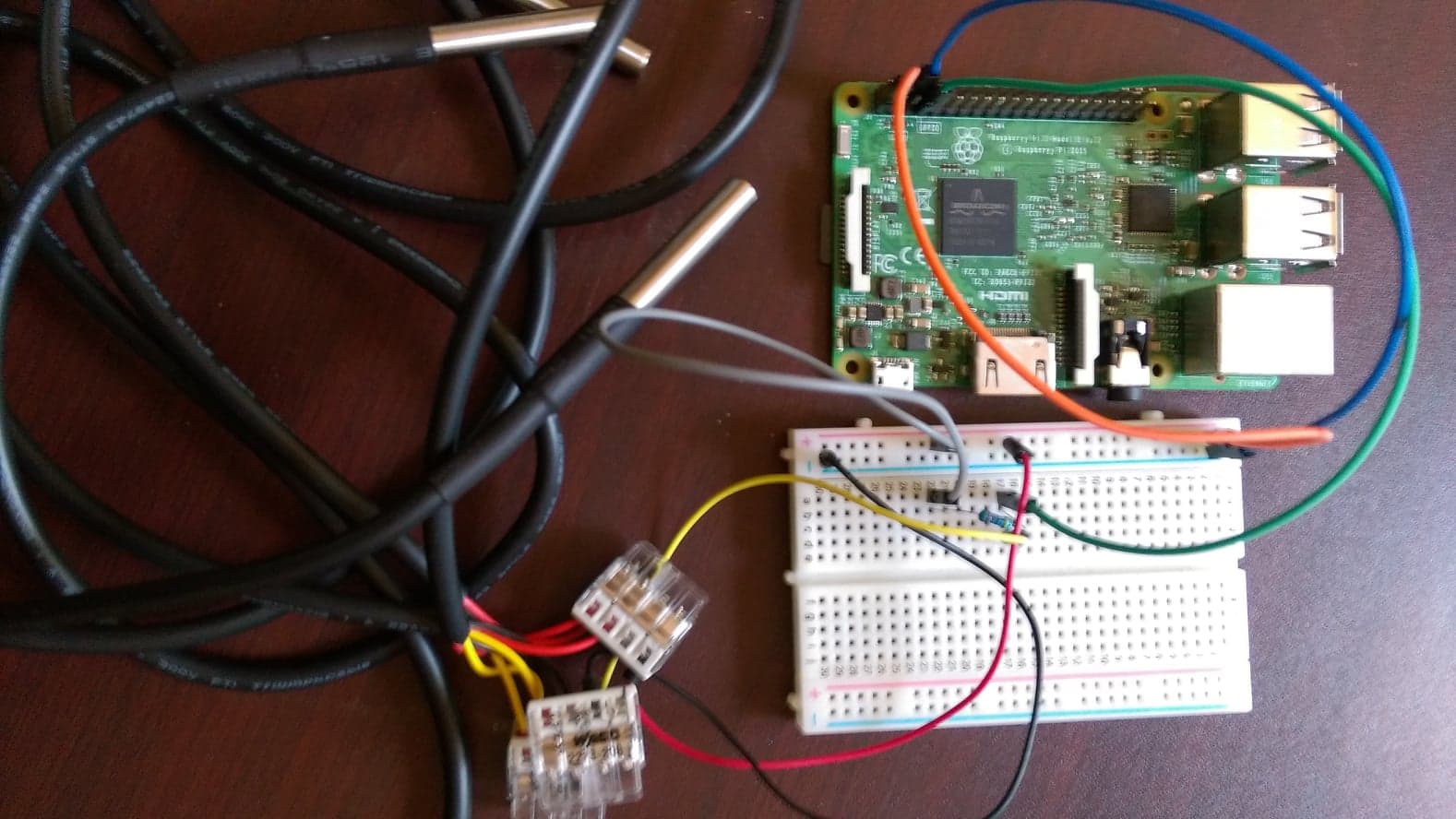


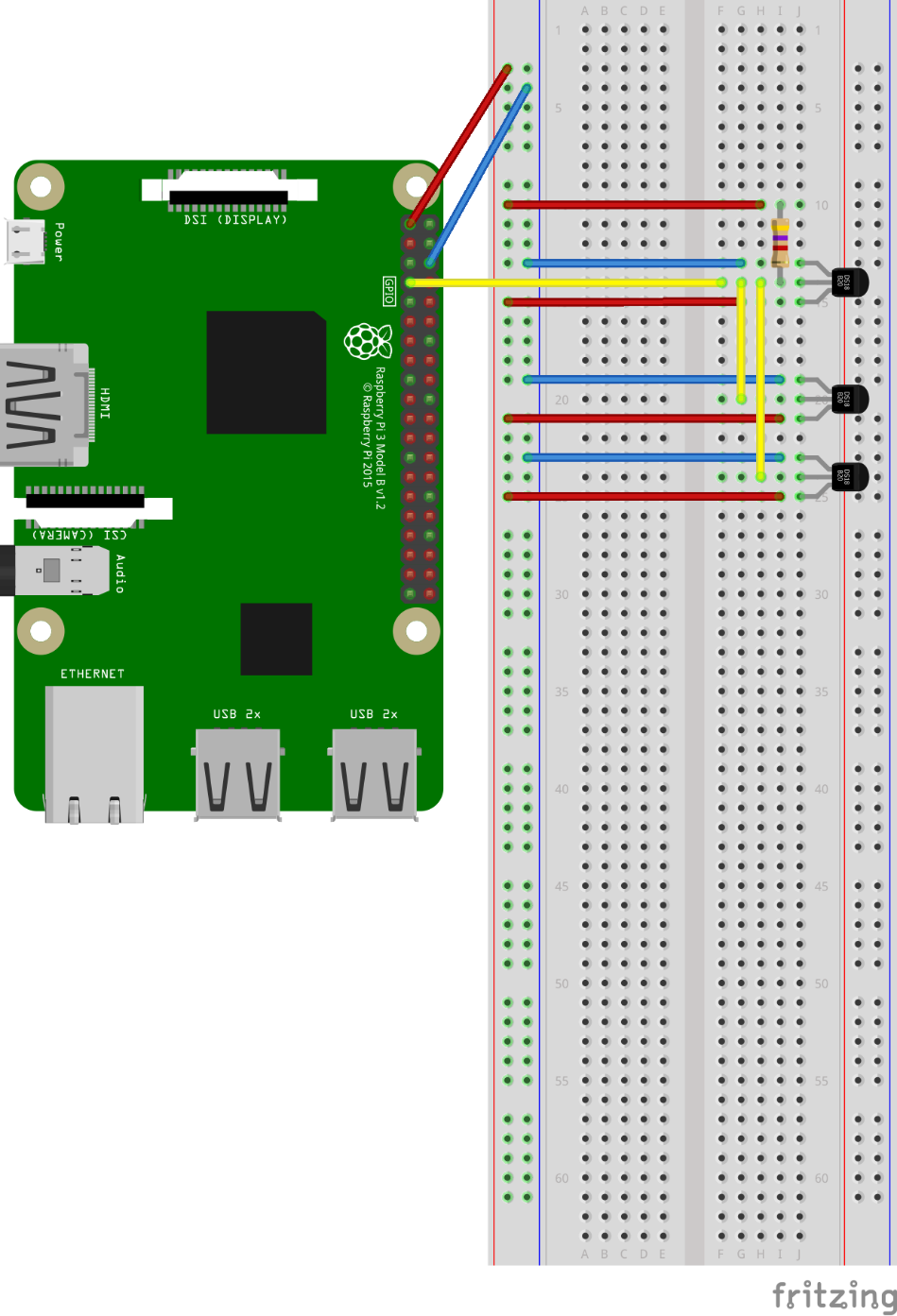


When we check the connection with Wireshark we can see that the packages are encrypted with TLSv1.2 and that we send the data on the secure-mqtt port 8883.



EXTRA: INSTALLING 1-wire sensors



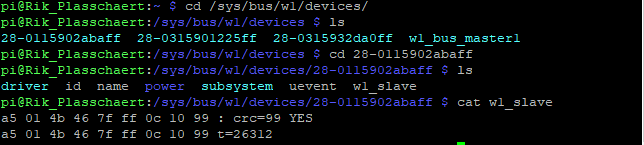


Al sensors are connected together and use 1 mutual data, vcc and ground pin. The pi delivers the power (3.3v) to the sensors on via the bus. The common data pin is connected to GPIO4. Between de datapins there is a 4.7K pullup resistor. This is used as a safety method to ensure that inputs to the pi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

To see wich and how many 1 wire snesors 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:

If we want to use these sensors to collect data the best way is to create a class. Then we assign functions to this class which we need in order to use the sensor. You have to make sure you keep this script in the same directory where you put the other scripts that make use of the class DS18B20.

*# Findingsensors.py*

*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))*

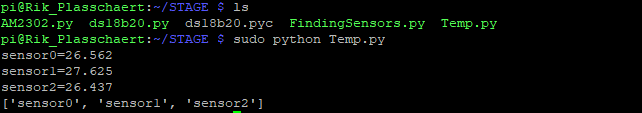
*device\_list.append('sensor'+str(i))*

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

*i += 1*

*print device\_list*

OUTPUT:



For the time being we only read out the sensors 1 time and print their value on the screen. Later on we will make sure we keep reading the sensors and transport their data to the database.

BANK PDU MONITORING

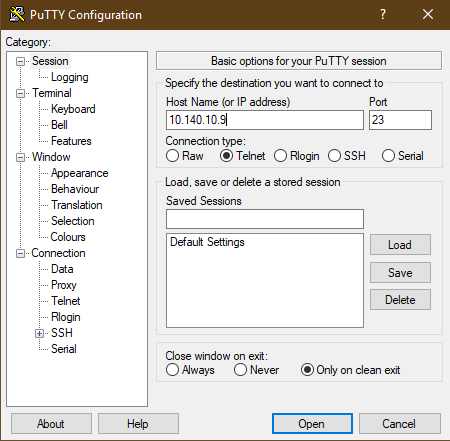
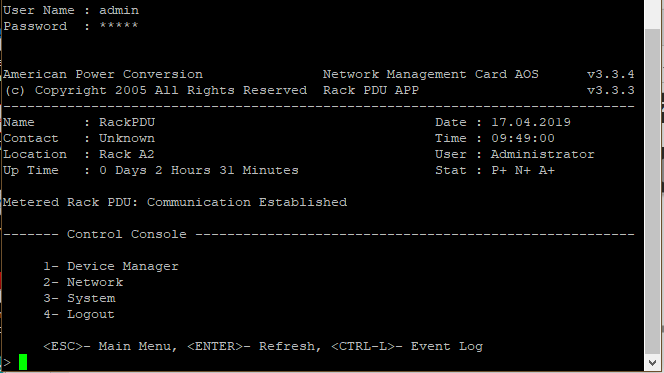
Finally we also want to measure the Rack PDU to which the switches are connected to. This way we can monitor the power consumption of the rack as well as the load status of the banks on the rack. PDU level monitoring is useful to determine whether no phase / bank is being overloaded so that certain equipment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

User Guide

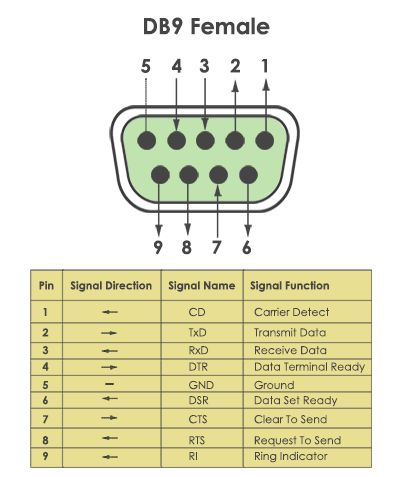
Metered Rack Power Distribution Unit

AP88XX

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 encountered some problems doing this 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.

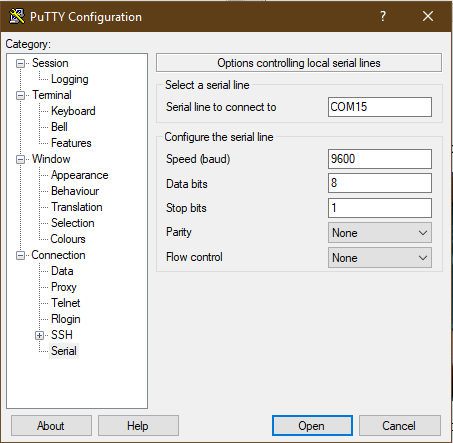
You will need to connect the wires accordingly:

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:



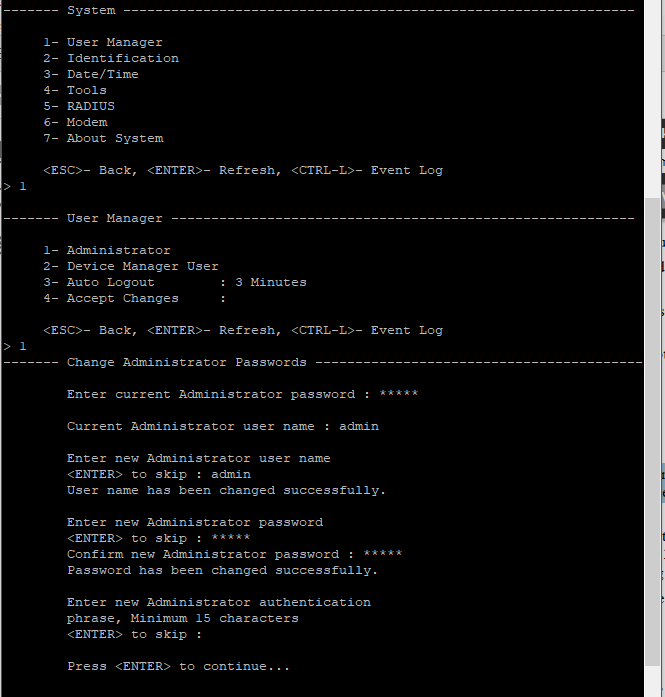
U need to find the correct port to wich the PDU is connected. Locate this port in Device manager on your pc. It is possible that you need to install a driver for the USB to Serial convertor.

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

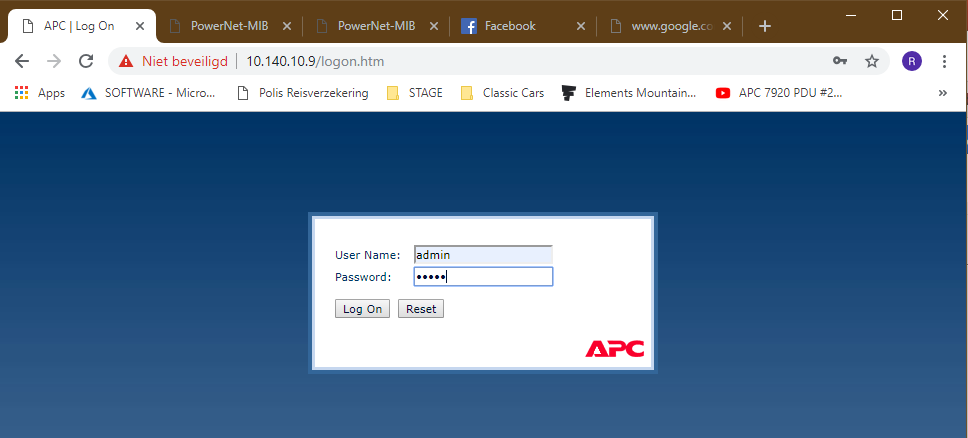
button a second time immediately while the LED is flashing to reset the user name and password

to their defaults temporarily.

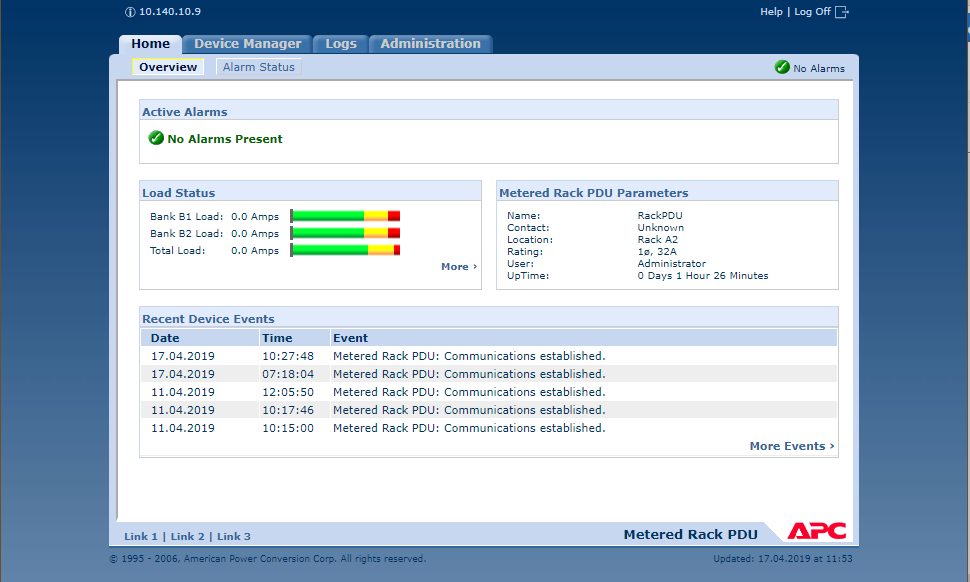
Press enter again to display the User prompt. Then use the default, apc, for username and password. You should be able to login. Once you entered the console you can find your way to the accounts and change the username and password to whatever you choose.



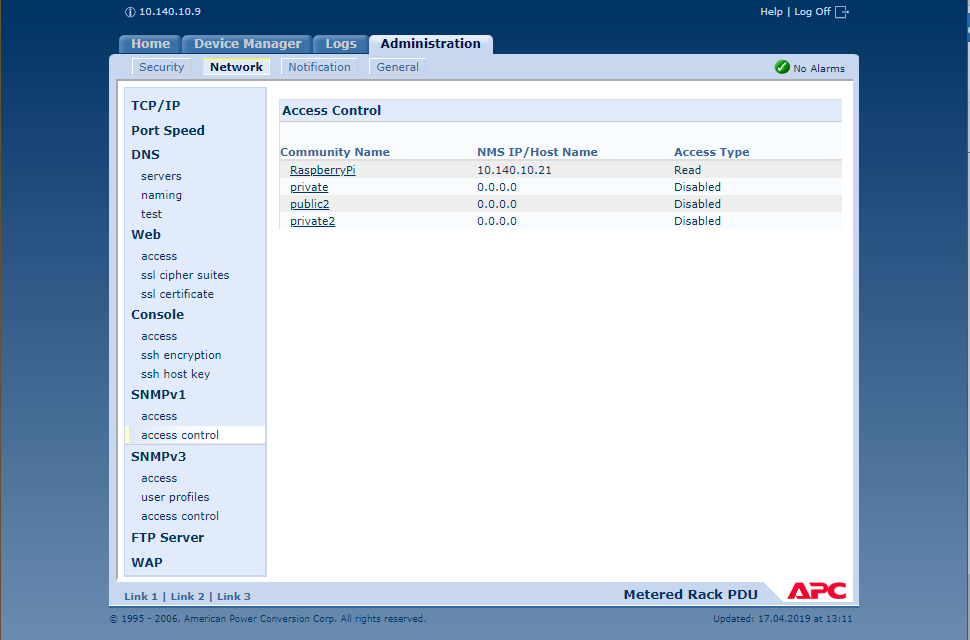
You can also login using the webinterface. Go to the ip address of the PDU and enter the correct credentials.



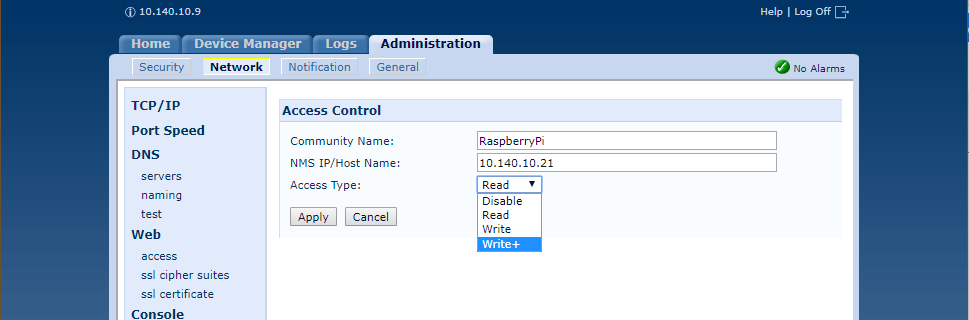
Here you can see the basic information of the rack. it is also possible to configure alarms, change network settings, manage users and many more.



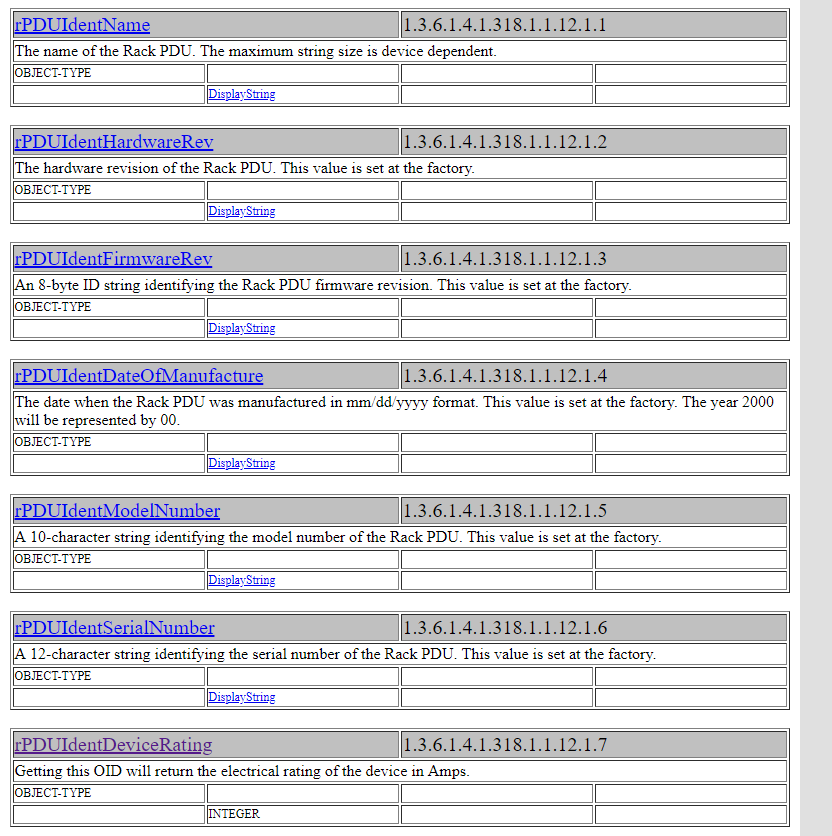
In order to make it possible for the raspberry pi to perform queries on the PDU we need to configure the PDU to allow read access from the pi. In order to do this go to the administration tab and select the Network tab. Here you see on the left side an list with possible network features to configure. 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.



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 particular device using OID’s. OID’s are identifiers and determine a path to a certain value holding information of a particular part of the device. We can use the Powernet MIB as a library to search for al the OID’s that are of value to us. Here is a example:



To test whether a 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 a number of 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.

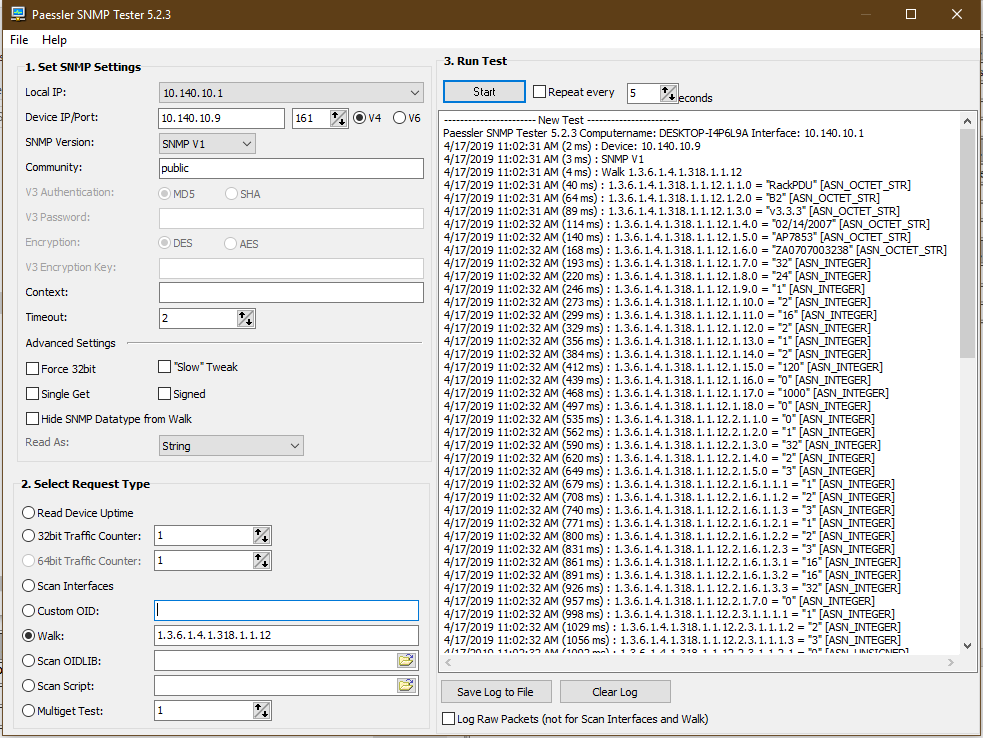
<http://www.circitor.fr/Mibs/Html/P/PowerNet-MIB.php>

We are working with old PDU’s (AP7853) and therefore 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

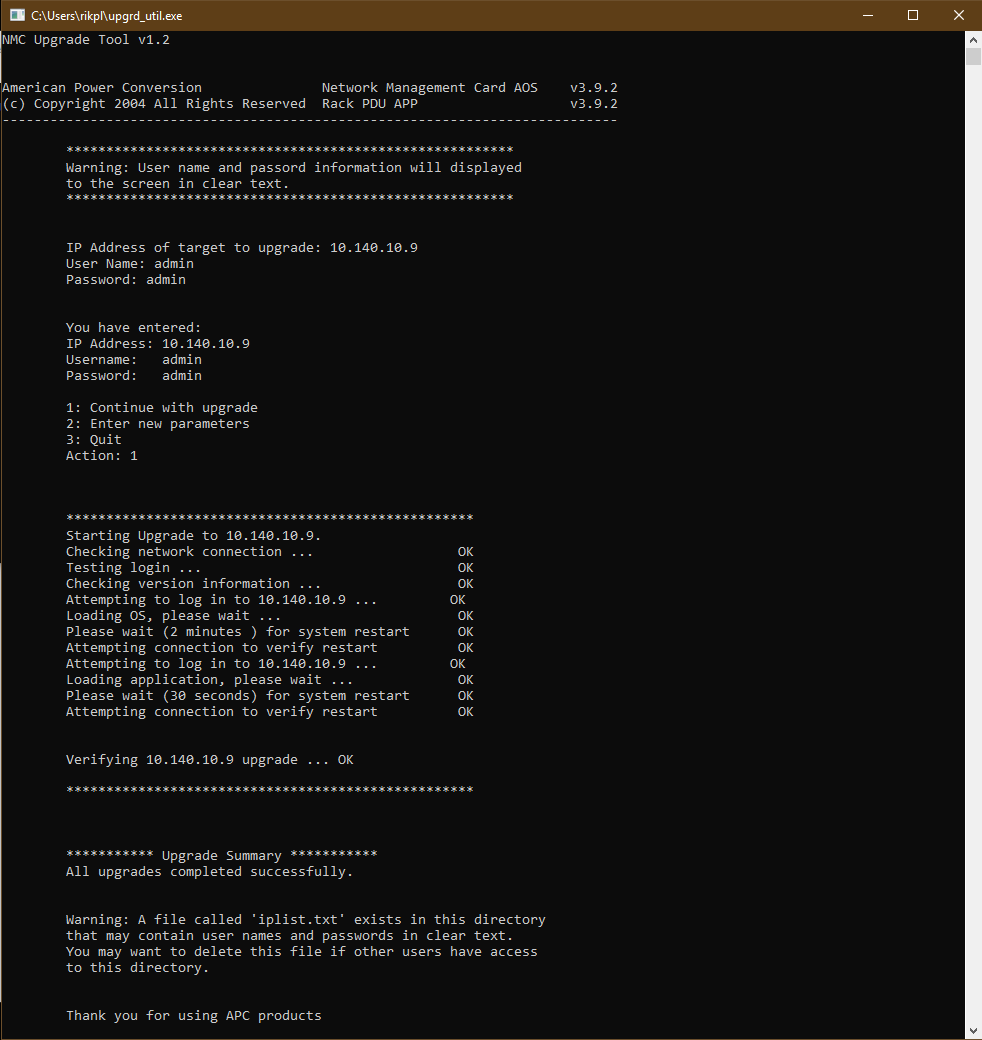
rPduStatusload 1.3.6.1.4.1.318.1.1.12.2.3.1.1.2/ .1 / .2 / .3

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



Ook nog checken bij andere pdu’s of bankcurrent werkt.

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.



SNMP

1.3.6.1.4.1.318.1.1.12.1

PHasecurrent – niet mogelijk

Bankcurrent – niet mogelijk

|  |  |
| --- | --- |
| [rPDUPowerSupply1Status](http://www.circitor.fr/Mibs/Html/P/PowerNet-MIB.php#rPDUPowerSupply1Status) | 1.3.6.1.4.1.318.1.1.12.4.1.1  powerSupplyOneOk(1), powerSupplyOneFailed(2) |
| [rPDUStatusBankState](http://www.circitor.fr/Mibs/Html/P/PowerNet-MIB.php#rPDUStatusBankState) | 1.3.6.1.4.1.318.1.1.12.5.2.1.3  bankLoadNormal(1), bankLoadLow(2), bankLoadNearOverload(3), bankLoadOverload(4) |

devicePowerwatts - 1.3.6.1.4.1.318.1.1.12.1.16

rPduStatusload 1.3.6.1.4.1.318.1.1.12.2.3.1.1.2

bepalen of er geen enkele fase/bank te zwaar belast wordt zodat bepaalde apparatuur niet langer redundant gevoed zou zijn.

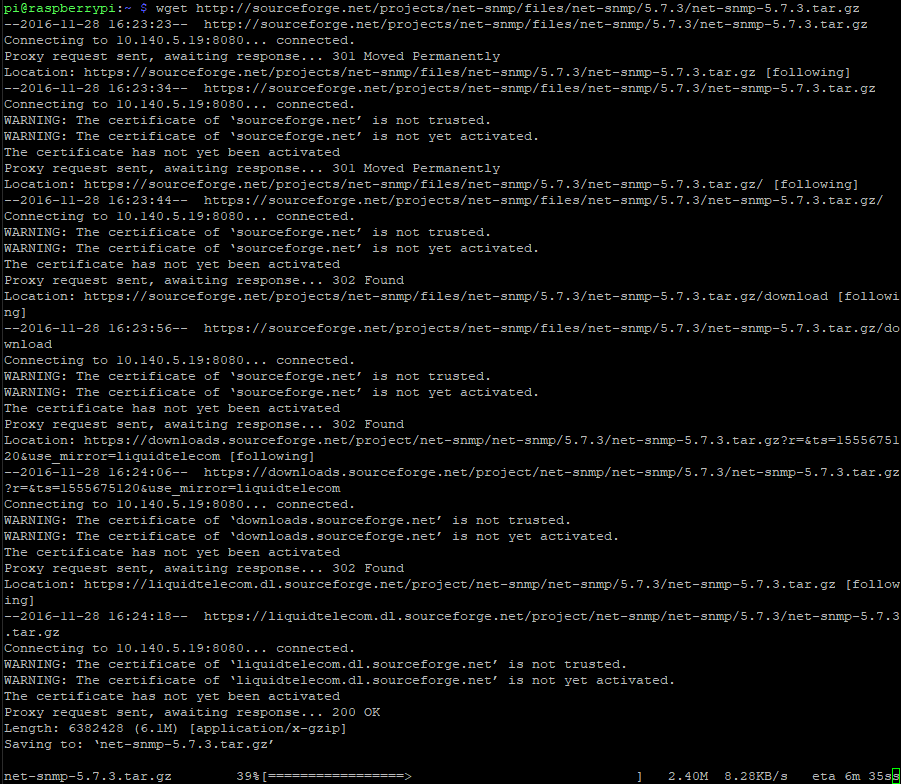
Install SNMP on the Pi

First you will need to download net-snmp using:

wget <http://sourceforge.net/projects/net-snmp/files/net-snmp/5.7.3/net-snmp-5.7.3.tar.gz> --no-check-certificates

we use the no-check because the certificate from soundforge is’nt 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:



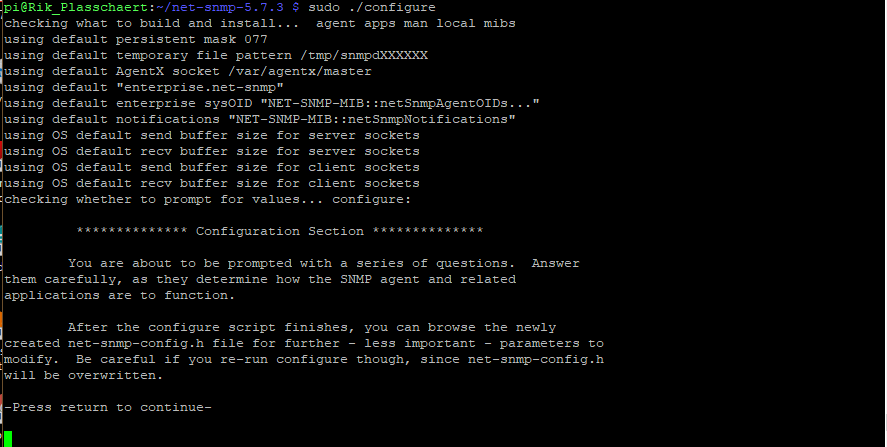
after this is done we can extract the downloaded file using:

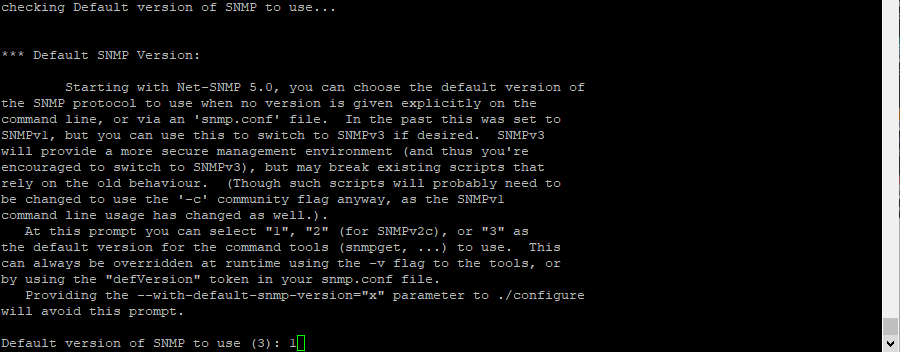


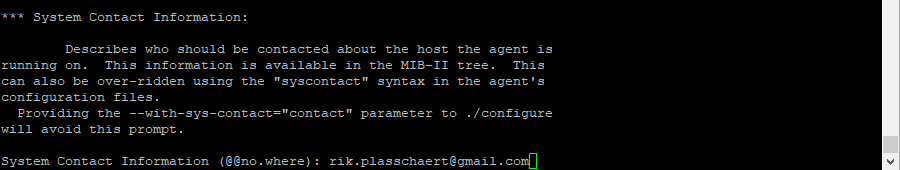
then we enter into the main directory and configure the package:

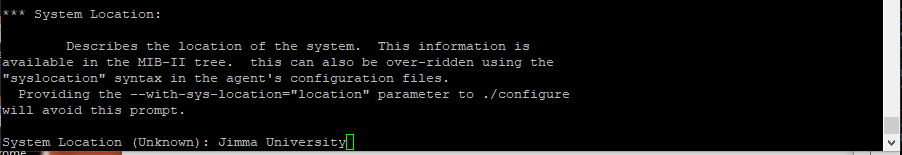


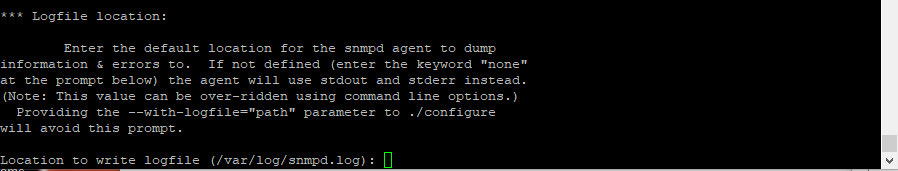
here we will need to fill in a few queries:

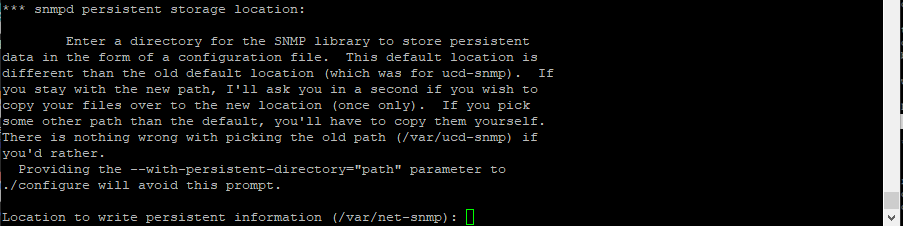


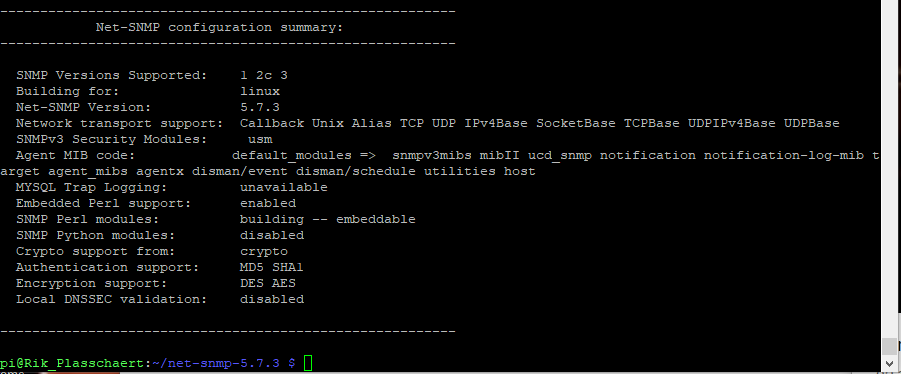










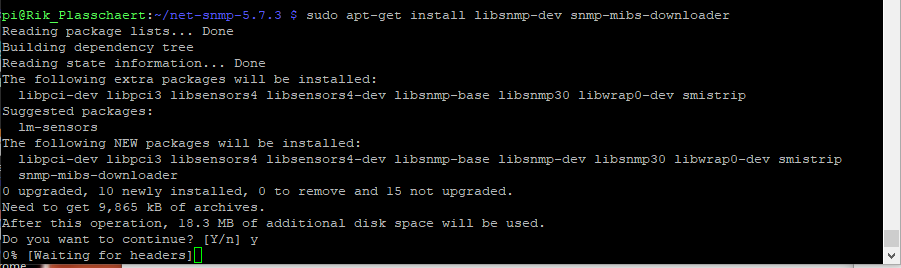
In the end you should see a similar summary. For the moment python modules are disabled, we will enable those in one of the next steps.

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

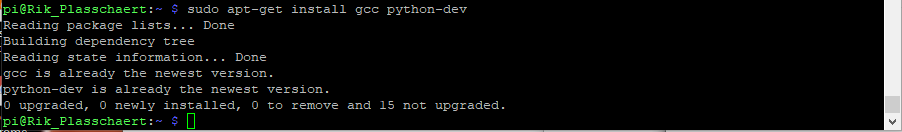




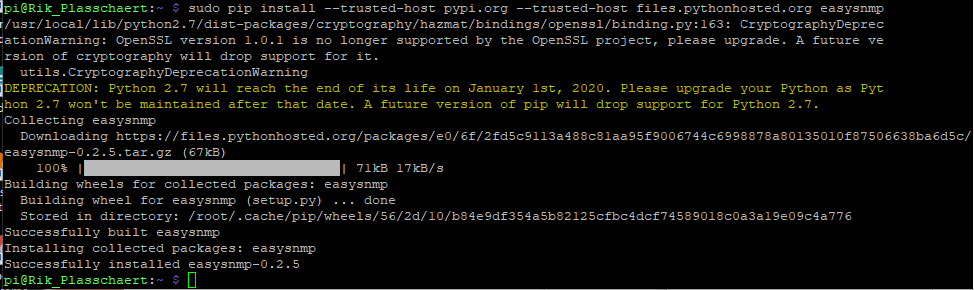
After this we install the easy snmp library and we download and install the MIB on the system:



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:

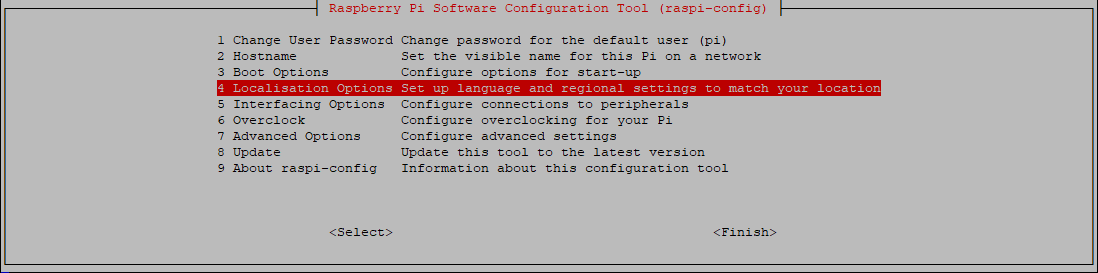
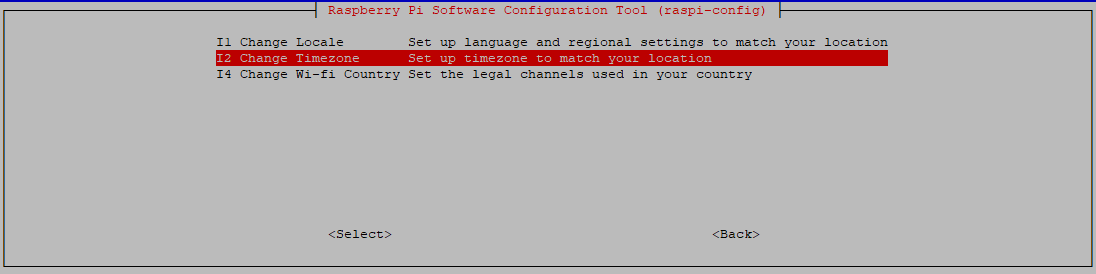


\*UPDATE

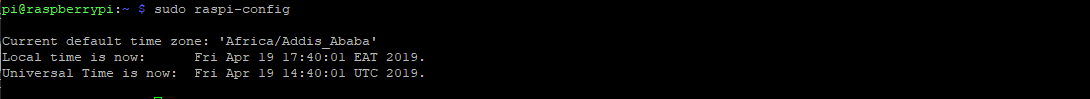
I could be that the errors occurred because the System time of the pi was wrong. 

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



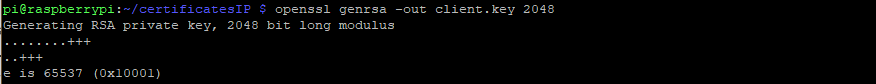
Before you do so make sure that the Pi’s time zone is correct. This project was completed in Jimma, Ethiopia. Enter sudo raspi-config on command line. Then choose next option:  

After you choose your location the following should appear:

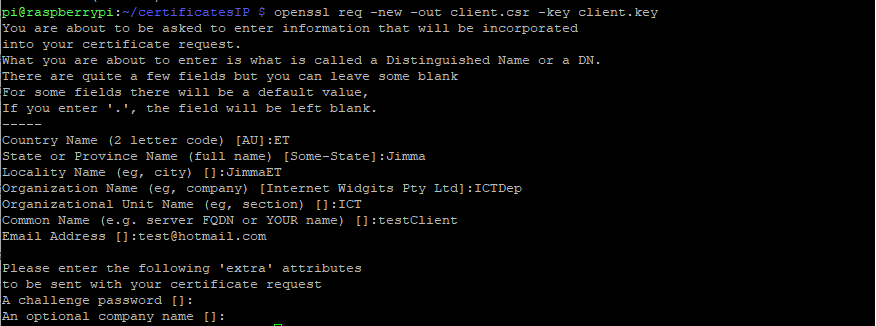


We can add extra authentication by using a client certificate. Here is how to create this:

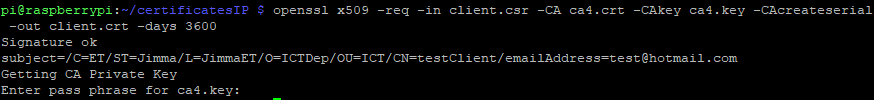
First we create a client private key:

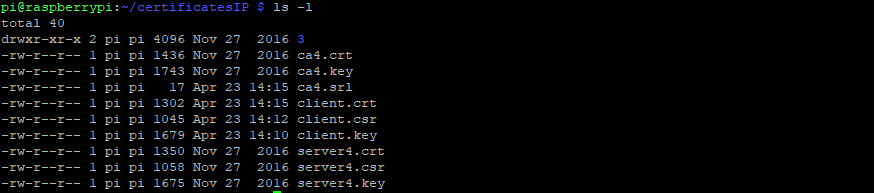


Next we create a certificate request and we use the client private key to sign it. The most important field is the common name field. Here we enter the name of the client so that the server knows who he is talking to.



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



Now the directory will look like this: 

If you use Python you need to set the TLS settings  using:

client.tls\_set('c:/ssl/ca.crt','c:/ssl/client.crt',\

'c:/ssl/client.key')