DVA 243

Data communications for Embedded Systems I

**Assignments INL1 report: Parts A, B, and C**

Students:

Adam Sandström

asm17005

[asm17005@student.mdh.se](mailto:asm17005@student.mdh.se)

Joakim Säteri

jsi17002

[jsi17002@student.mdh.se](mailto:jsi17002@student.mdh.se)

*Mälardalen University*

*Spring 2019*

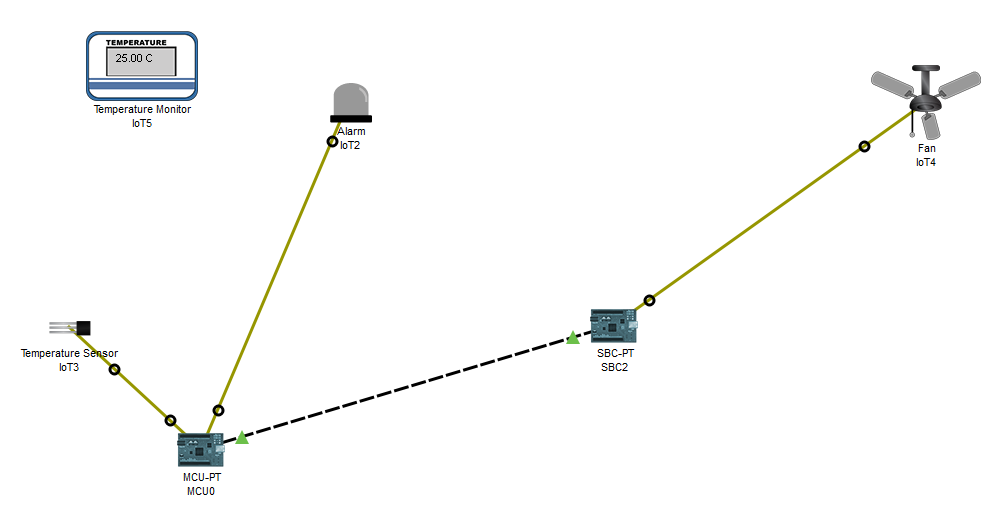
**Part A: Packet Tracer simulations**

In this part of the report you can read about our simulation we did to figure out the difference between time-triggered, event-triggered and mixed solutions for our system. And what solution we decided to use in our system.

**A short description of the goal of the assignment and the control system being developed in Part A:**

Our goal with the packet tracer simulation where to find the most suitable solution for the communication protocol between the raspberry pi’s (RPi).

We implemented three different solutions, one purely time-triggered solution, one purely event-triggered solution, and another one that was a mix of both the time-triggered solution and the event-triggered solution.



*Image 1 - packet tracer topology*

**Description of the tested solutions for the communication protocol:**

In the packet tracer simulation we implemented three different solutions, one was purely time-triggered, one was purely event-triggered, and the third was a mix of both time and event triggered.

**Protocol 1 – Time-triggered solution**

Our time-triggered solution was implemented using two nodes or raspberry pi’s, one node acted as both an actuator and a sensor consisting of a temperature measurement device and a light emitting diode (LED). The other node was purely a actuator only having a fan connected to it.

The sensor node read the temperature, converted the input data to celsius, and sent the data to the actuator node. If the measured temperature was above 26 degrees celsius it would turn its own LED on otherwise it would turn it off. After the sensor node had finished all its tasks it would sleep for one second before doing them all again in an infinite loop.

The actuator node ran an udp server on a different thread than the main thread. Packet tracers udp library does this automatically in the background. In the main thread we ran an infinite loop with a sleep for five seconds, this serves no real purpose, we only implemented the sleep to have something in the loop so the program would keep running.

**Protocol 2 – Event-driven solution**

The event-driven solution continuously reads the temperature data from the thermometer and compares the input data it receives with two different conditions.

The first condition states that if the input temperature is greater than or equal to 26 degrees celsius and the LED and fan are turned off, the node will turn the LED on and send a signal to the actuator to turn the fan on.

If on the other hand the input temperature is lesser than 26 degrees celsius and the LED and fan are turned on the sensor node will turn its own LED off and send the signal to the actuator to turn its fan off.

A two-way handshake was also implemented to handle packet drops. The two-way handshake solution was implemented by having the actuator node send a message back to the sensor node every time it receives a udp message from the actuator node. The sensors node will continuously send the signal to either turn the fan on or off in a loop with a one second sleep until the sensor node receives a message back from the actuator node. When the sensor node receives a message from the actuator node it will change a variable in its python code to true, indicating the actuator node has received the data, this will result in the sending loop to break and the sensor node to stop trying to send the data to the actuator node, in effect a confirmation that the data has been delivered.

**Protocol 3 – Mixed solution**

The mixed solution was built on the same communication protocol as the event-triggered solution with the addition of a hello message being sent every five seconds from the sensor node to the actuator node.

When the actuator node receives a message it checks the contents of the received udp message to separate out the hello-message from the messages containing the temperature data, this is so that the two-way handshake and the hello-messages will work without the sensor node requiring a confirmation for every hello-message.

The purpose of the hello message is to make sure the actuator knows if the connection gets dropped to the sensor node. A further potential implementation to this solution would be to either have the fan start or trigger some kind of alarm if the connection is lost.

**Description of Packet Tracer simulations and the achieved results:**

We saw some strengths and weaknesses in all protocols after we did the packet tracer simulation.

**Time-triggered solution**

The strengths of time triggered is that the actuator know that the sensor is alive, since it keep getting messages. The weakness is that it sends a lot of data that doesn't need to be sent. Another thing is that it can be hard to handle packet loss, if you don't want to send a lot of data, you need to keep the time between messages longer, which give less time for handle errors that comes in bursts. If you want to have more time to handle packet errors you need to lower the time between sent messages but this will result in more data over the network.

**Event-triggered solution**

The strengths of event triggered send data when an event happen, that makes it fast and we think it's easier to handle errors in this kind of system and still make the deadline. Event - triggered by itself is not that good in handling packet errors, since it only sends a packet when something happens, this is why we thought it was needed to implement acknowledgements. No unnecessary data is sent. The weakness of event triggered is that the actuator has no idea that the sensor is still alive.

**Mixed solution**

The strength of the mixed solution is that it sends the data when it needs to be sent, that is in an event-triggered fashion. It also has the strengths of time triggered, because the actuator will be able to tell if the sensor is still alive. The mixed solution keeps the weakness of the time-triggered solution since it sends more data than the event-triggered solution, but we think it's worth it so we actually know that the sensor is still alive incase it would be a long time before an event happens.

**Final protocol selection:**

We chose to use the mixed solution for our real implementation because we didn’t know how sensitive the system should be to packet-drops and this solution seems to us to be the most safe solution. For the mixed protocol we used a 5 second interval for the hello messages the reasoning for this is that we thought that 5 seconds would be a good time interval to check if the connection was still active.

The event that would trigger the sensor node to turn the LED on and send a signal to the actuator node to turn its fan on was if the sensor node measured a temperature greater than or equal to 26 degrees celsius and the LED and fan being turned off, meaning it would not try to turn the LED on or send a signal to the actuator node telling it to turn the fan on if the fan and LED was already on and thereby reducing the total amount of data sent over the network.

The event that would trigger the sensor node to turn its LED off and send a signal to the actuator node to turn its fan off was if the sensor node would measure a temperature lesser than 26 degrees and the LED and fan being turned on. The reasoning for this was the same as the reasoning to why we wouldn’t want to send the data to turn the LED and fan on if they were already on, that is to reduce the total amount of unnecessary data being sent over the network.

We chose to implement the control on both nodes but the sensor node was the most active node since this node was reading the temperature and making the decision to actually send the data or not. The actuator node was more passive but it still analysed the data to make sure it was either equal to or above 26 degrees celsius to turn the fan on or below 26 degrees celsius to turn the fan off. We also needed the actuator node to analyse the contents of the received messages to handle the two-way handshake and the hello-messages.

To handle packet loss we implemented a two-way handshake where the sensor node would keep sending the data to either turn the fan on or off until it received a message back from the actuator node. For this to work with the hello-messages we had to make sure the actuator node didn’t reply to the hello-messages.

A keep-alive function was implemented by having the sensor node send hello-messages every five seconds to the actuator node. This would be improved on in part b to have the actuator notify the user of a lost connection if it didn’t receive a hello-message in a certain time-interval, but this alert function was not implemented in the packet tracer solution, only the sending of the hello-messages was implemented in packet tracer.

Since our two-way handshake will continuously keep sending the signal to either turn the fan on or off until it receives an acknowledgement from the actuator we believe our solution will work no matter how many packets gets dropped.This mean that our system is still functioning as long as the errors don't last as long or longer than the deadline. One problem with this solution in packet tracer was that it only looked for the hello-messages in the received messages where it should have also looked at the content and if something that it wasn’t expecting were received it shouldn’t send a message back. So our packet tracer solution isn’t safe against packet errors. In part b we improved on this.

# 

**Part B: Implementation in real hardware**

In this part of the report we will take a look at our system that we developed. How it works and what each part of the system is responsible for. How things are connected.

**A short description of the developed system and its functionalities:**

The system consist of two Raspberry pi´s (Rpi) that are connected with each other with ethernet cables over a switch. One raspberry pi is using a thermometer to measure the temperature, this raspberry pi is the sensor. The second raspberry pi has two led lights connected to it, this one is the actuator. If the temperature in the room is measured to be 26 degrees celsius or higher and the led is turned off, the sensor will send a message to the actuator and tell it to turn one led on. If the temperature goes below 26 degrees celsius and the led is turned on the sensor node will send another message to the actuator to tell it to switch its lights off. If the actuator does not receive a message to turn the light off within 5 seconds it will turn its second led light on. The sensor also sends keep alive messages every 2 seconds. If the actuator hasn't received a keep alive message within 6 seconds it will send a warning message to the screen indicating that the connection is down.

**Technical documentation:**

We made some changes from the design we’d chosen after part A. Instead of sending temperature data over the network we decided to just send different numbers instead. This was implemented so that it would be easier to see if the data had one of the expected values. Instead of the temperature we decided to let a number 1 represent that the temperature has gone up to 26 degrees celsius or higher, a 0 would represent that the temperature has dropped below 26 degrees celsius and that the led should be turned off. We used the data value 2 to represent keep-alive messages. If the connection would be lost, that is if we don’t receive a keep-alive message within 6 seconds an error message will be printed to the screen. This could be expanded upon to to perhaps trigger an alarm.

**The source codes:**

The system has two source files that are written in python, one for each raspberry pi. The source file for the actuator is called aktuator\_node.pyand the source file for the sensor is called sensor\_node.py. (names may differ in the added screenshots in this report, since the name changed on these source files under development).

**The actuator source code:**

The python script called aktuator\_node.pyis responsible for handling incoming messages from the sensor, and to send acknowledgements back that it received a message.The acknowledgement would contain either a 1 or a 0 depending on what kind of message it acknowledged, this was implemented so that we can identify that the actuator received the correct message. The actuator is also responsible for turning the leds on or off depending on what information there is in the message received. The actuator also checks if the connection is established between the two by checking if it has received a keep alive messages within the previous 6 seconds. The deadline for keepalive messages was set to 6 seconds incase one or two keep alive messages would be dropped there is still time for a third message to arrive before it triggers a warning about the lost connection to sensor.

**The sensor source code:**

The sensor\_node.py is the sensor node and is responsible for checking the thermometer and calculate the value it receives from it to celsius and check if the temperature received is 26 degrees celsius or higher. If the temperature is 26 degrees celsius or higher and the leds are currently turned off it will send a message to the actuator node (the computer running the [namn34] script) telling it to turn the led on. If the temperature on the other hand would rise above 26 degrees celsius and but with the led already being turned on the sensor node would not send the signal to turn the led on. This signal is a message with its data value set to 1.

**How it works together:**

If the sensor would measure the temperature to be below 26 degrees celsius it would either send the signal to turn the led off if it is on or do nothing if the led is already turned off. This signal is a message with its data value set to 0.

Every two seconds the sensor will send a keep alive message to tell the actuator that it's still alive and actively checking the temperature. The thermometer has its own file in linux, that the source code will load in to working memory while executing the python script. If the source file is used with another thermometer there will be a need to change the source code for the sensor, since every thermometer has its unique serial number, which is indicated by the file name being loaded into memory by the script.

In the event of a temperature reading 26 degrees or higher the sensor will start sending a message containing the data value of 1 to the actuator until it has received an acknowledgement back with the same data value from the actuator node.

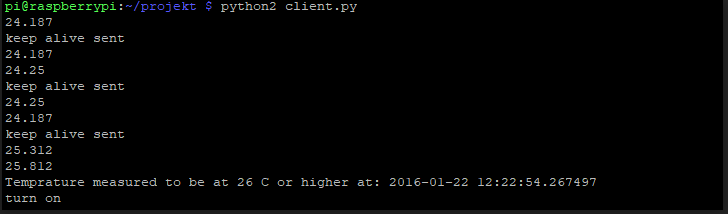
The acknowledgement should also contain a message containing a 1, then the system knows that it was an acknowledgement for the correct message.

If the temperature measured is below 26 degrees celsius and the led is currently turned on a message with the data value set to 0 will be sent to the actuator node until it receives an acknowledgement message with its data value set to 0 from the actuator node.

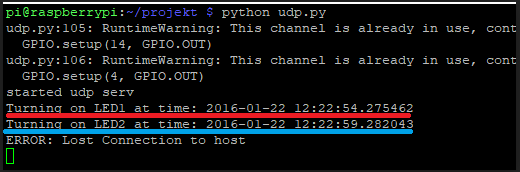
For the keep-alive messages there is no acknowledgements. If the actuator or the sensor receives a message that's not intended it will ignore it.

**Time synchronization**

The actuator is also working as a ntp server for our system, and the sensor is acting as a client. This was made so we could get our time synced to see how long time it took for the temperature to be measured and then sent and received and finally turn on one led. (see picture 2 and 3 below for the time measurement).

**

*Image 2. The image shows what time the sensor have read the temperature to be at 26 degrees celsius or higher.*



*Image 3: The image shows what time the actuator is turning on the leds.*

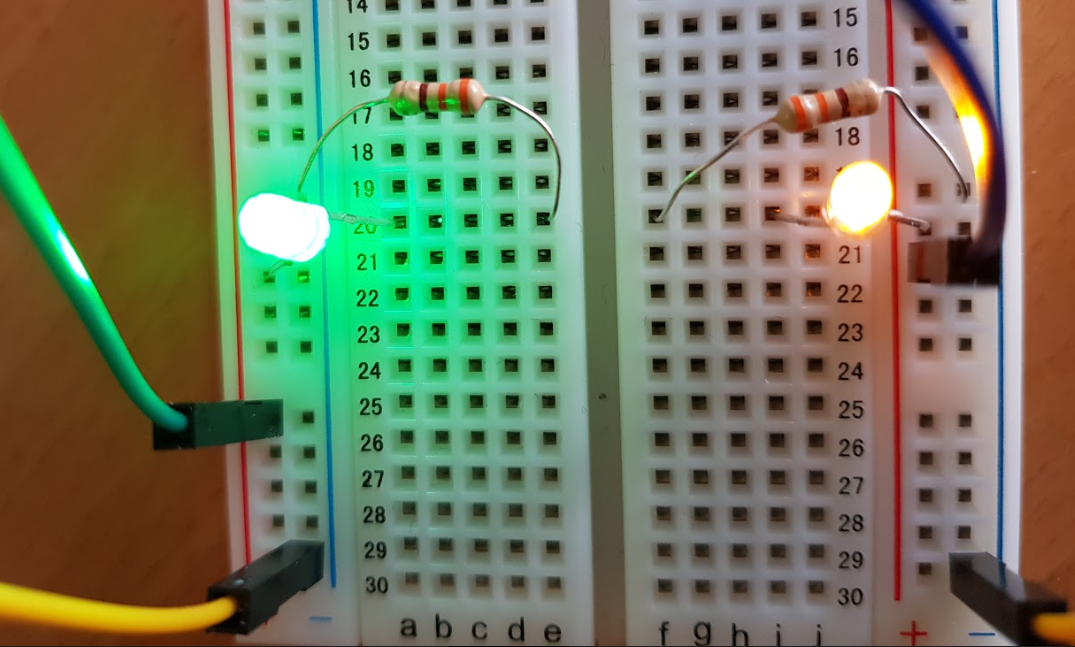
**How to set up the hardware**

Below are descriptions with images on how to connect the RPi’s to the breadboards with the LEDs, thermometer, and resistors.

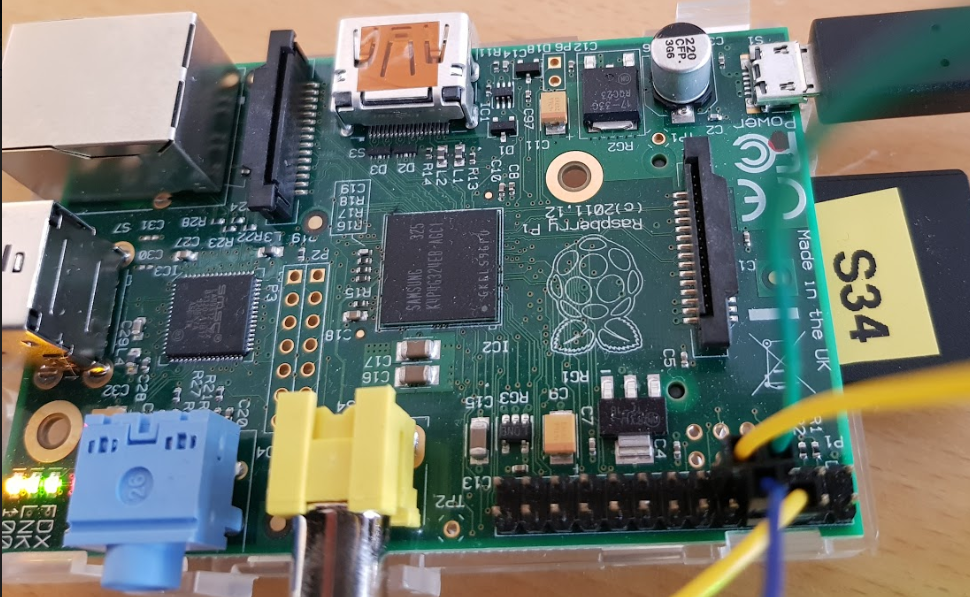
**RPi (S34) Actuator node**

First connect the negative jumper cables as described in images below. Then on the breadboard connect them anywhere on the negative columns marked with a blue line and the negative (-) sign as described in images below. For the positive cables (blue and green) connect them on the RPi as described in images below. On the breadboard connect them anywhere on the columns with the red line and the positive (+) sign.

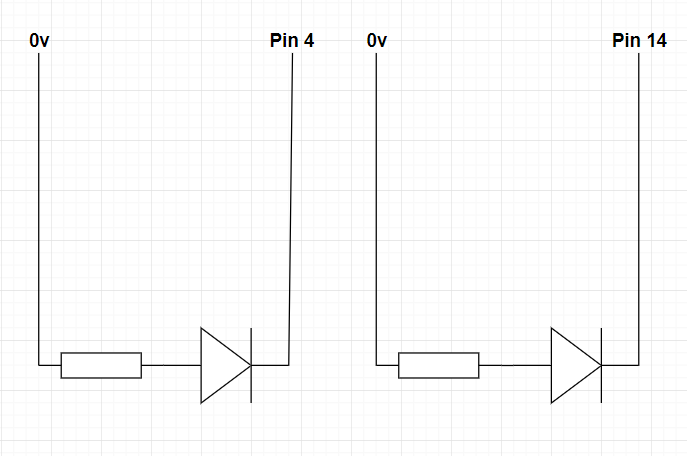
For the resistors connect one end anywhere on the negative column (blue line) as in image 2. The other end can be connected on either of the rows marked with numbers ranging from 1 to 30. From the same row connect the short end of the LED and the longer leg of the LED anywhere on the positive (red) column.



*Image 4 - actuator connections*



*Image 5 - Actuator GPIO pins*



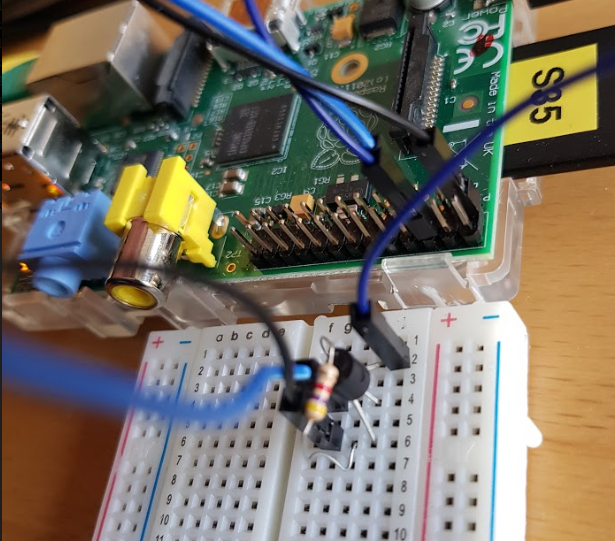
*Image 6. Picture shows a circuit diagram for the actuator, it shows a resistor, a led and what pins on the rpi to connected it to.*

**RPi (S35) sensor node**

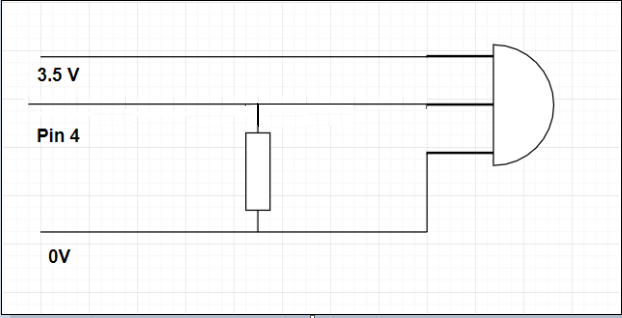
For row 2 of the breadboard connect the dark blue cable to the row and the other end of the cable to one of the negativa pins on the RPi, also connect the first leg of thermometer to the second row of the breadboard (see image 7 and 8).

For row 4 of the breadboard connect the middle leg of thermometer, the lightblue cable from GPIO pin 4 on the RPI, and one leg of the resistor (see image 7 and 8).

For row 6 of the breadboard connect the third leg of the thermometer, the other leg of the resistor from row 4, and the black cable from the +3.3v pin on the RPi (see image 7 and 8).



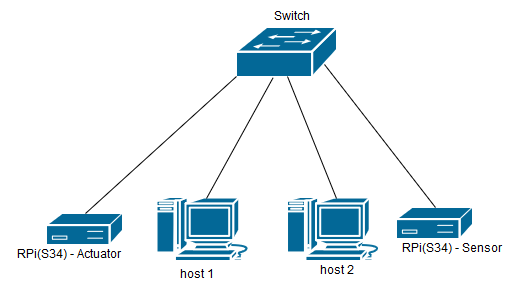
*Image 7 - Sensor RPi and connections*

**

*Image 8 - The image shows a circuit diagram for the sensor, its shows a resistor and a thermometer and what pins to connect it to on the rpi.*

**Network topology**

The network topology we used for this is described in image 9 below.



*Image 9 - network topology*

We connected all the hosts (the actuator, host 1, host 2, and the sensor) with ethernet cables to a switch. As described in image 9 we didn’t use a router since we didn’t need to reach outside our local area network for this to work. Even though we never connected a router we still statically assigned the ip address for a gateway on the raspberry pi’s so that if we wanted to connect one we would only have to assign its ip address to 10.0.0.1/8.

**The ip addresses of the Rpi’s**

|  |
| --- |
| Rpi(S35) sensor node  IP: 10.35.0.2  Subnet mask: 255.0.0.0  Gateway: 10.0.0.1  Rpi(S34) actuator node  IP: 10.34.0.1  Subnet mask: 255.0.0.0  Gateway: 10.0.0.1 |

**The ip addresses of host 1 and host 2**

These can be assigned to any ip address in the 10.0.0.0/8 network except the once already in use. We chose to assign them the follow ip addresses.

IP: 10.10.0.1

Subnet mask: 255.0.0.0

Gateway: 10.0.0.1

Rpi(S34) actuator node

IP: 10.10.0.2

Subnet mask: 255.0.0.0

Gateway: 10.0.0.1

**Starting the system**

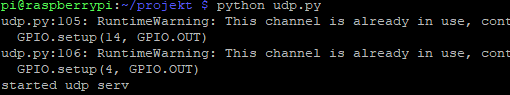
Once all the host are connected as described in image 5 we can connect from the PCs (host 1 and host 2) to the raspberry PI’s with ssh.

Once connected to the raspberry PI navigate to the ~/projekt folder. You can use this command to move to that directory:

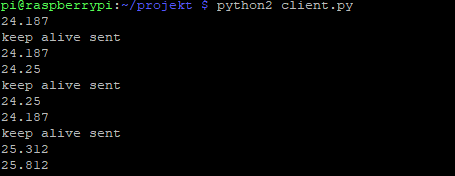
|  |
| --- |
| cd ~/projekt |

Inside of the folder there should be either a file called udp.py (our source code files have different names on our memory card, we forgot to change them) on the actuator node or client.py on the sensor node. To execute the files run them in python with the command as described in image 10 and image 11.

A sample output of the programs in action are shown in image 10 and image 11. The warning message in image 10 can be ignored it’s only there because we didn’t shut down the actuator node properly.



*Image 10 - running the actuator node python script.*



*Image 11 - running the sensor node python script.*

**Part C: Functionality assessment**

In this part of the report we will discuss how we tested our system with packet errors and how it could handle it. We will also take a look one some other errors that could occur and how we think our system would handle it.

**Description of the introduced errors:**

To test the robustness of our system we implemented code to simulate packet errors. We made two copies of the original script, one for each error to be simulated. The first script will randomly send a packet to the sending host simulating a dropped packet and the second script will send a burst of 10 packets to the senders address simulating a burst of packet drops.

**Scenario 1 – randomly distributed errors**

To generate random packet drops we implemented the following code in the sensor node (RPi 35).

|  |
| --- |
| if count % randNum == 0:  s.sendto("5", adress2)  randNum = random.choice([3,5,7,10,13,17])  count +=1 |

This code will generate a random number from the numbers 3,5,7,10,13, or 17 and also increment the variable count by one for each packet sent. To simulate a packet drop we sent the packet to the address specified in the variable ‘adress2’ which holds the socket of the sender, meaning it won't be sent to the other node.

For every packet being sent the variable ‘count’ will be incremented by one. The first time running the script a value for the ‘randNum’ will be generated, and everytime a packet drop is simulated a new value for ‘randNum’ will also be generated.

The packet drop will be simulated everytime the number of sent packages indicated by ‘count’ is divisible by the value of the ‘randNum’, when this happens the system will try to send the next packages to itself simulating a packet drop.

**Scenario 2 – errors appearing in bursts**

To generate burst errors we made the program send ten packets to itself the first time we were about to turn the LED on, each time the program was run. The reason for this is to have packet drops happen when the data being delivered is actually important, which is hard to implement with the random packet drop errors. With this error simulation we are able to see how resilient our system is to errors. The burst error implementation was implemented by the following code.

|  |
| --- |
| if temp >= 26 and not isOn:  try:  while not answer == 1:  if burstTime:  for i in range(10):  s.sendto("5", adress2)  burstTime = False  else:  s.sendto("1", adress)  isOn = True |
| The for loop will send 10 packets with the data “5” to ‘adress2’ which is to itself to simulate packet drops. Data code “5” is just a random number used for these error test with no other special meaning. The sending of the burst errors are controlled by the variable *‘burstTime’* which can be set to True to make the program send 10 burst errors. We have implemented this to only send the first time the program is about to send the signal to turn the LEDs on on the actuator, but this can easily be implemented to have the burst error occur randomly during normal execution.  Since the random error implementation randomly drops one packet we couldn’t test the robustness of the system with random errors and this is our motivation to implement the burst errors when trying to send the on signal to the actuator. |

**Conclusion:**

Our evaluation of the errors we introduced, is that the code is smart enough to handle both types of errors, that is both random packet drops and bursts of dropped packets. How it handles this is with our implementation of a two-way handshake that keep sending the message until we receive an acknowledgement, meaning the system will keep trying to send the data until connection to the actuator node is established again, so that it will eventually get to the destination.

The only way we wouldn't get to the destination within the deadline is if we got burst errors that lasted for more than 2 seconds or some other loss of connection, then we would miss our deadline. A potential solution for this is to have the led or some kind of alarm system go off after not receiving keep-alives for a little less than 2 seconds. Or perhaps just have the LED turn on after not receiving keep-alives for 2 seconds, but since we didn't really know what this system will be used for, we didn't want to implement this. For example if it was an airbag in a car, and it went off just because it lost connection to something, that would be a stupid design decision.

That's why we implemented keep alives. The actuator has a timer on 6 seconds that resets every time a keepalive is received, and keep alive messages are sent every 2 seconds from the sensor. If we lose three or more keep alives in a row our system would consider the connection dropped and warn about a lost connection. This function could easily be extended to have an alarm go off or have the light turn on or something else suitable to the application.

We are also protected against corrupted data by having the sensor not send the temperature data but instead send certain codes we could reduce the number of valid inputs for the actuator node making it much easier for the actuator node to validate the received data. If the received data code doesn’t match any of the expected data code an error message will be printed, this can also be expanded upon to suit the intended application of the system. When the actuator node receives a packet with an expected data field it will send the received data back to the sensor node, until the sensor node receives this packet it will keep sending the packet, this makes our system safe against corrupted data, randomly dropped packages, burst errors, and loss of connection since the sensor will just keep trying to send the data until connection is established again.