

CSU33031 Computer Networks

Assignment #1: Protocols

Stephen Davis, Std# 18324401

October 30, 2021

Contents

1	Introduction.....	2
2	Theory of Topic.....	2
2.1	Internet Protocol (IP)	2
2.2	User Datagram Protocol (UDP)	2
2.3	Communication	3
2.4	Packet Description.....	3
2.5	Docker.....	4
3	Implementation.....	4
3.1	Node.....	4
3.2	Acknowledgements (ACKs)	5
3.3	Dashboard	5
3.4	Broker	8
3.5	Sensor (Publishers).....	10
3.6	Actuator (Subscribers).....	11
4	Discussion	12
4.1	My Design	12
4.2	Advantages of my Design.....	14
4.3	Disadvantages of my Design	15
5	Summary.....	15
6	Reflection.....	15

1 Introduction

The problem description for this assignment outlined protocol development, where the aim of the protocol is to provide a publish/subscribe mechanism for processes based on UDP datagrams. A publish/subscribe protocol is one where a publisher publishes messages to a broker, and the broker then forwards these messages to the corresponding subscriber(s). To support functionality for the protocol, one must look at the information that is kept in the header. The description defined that there are 2 basic requirements for the protocol: 1) the reporting of sensor data to a number of subscribers and 2) the issuing of instructions to actuators. This means that the functionality the protocol has to provide is the subscription to topics by nodes to a broker and the publication of information from nodes to topics at the broker.

In this report, I will discuss the theory behind the topic of this assignment, describe my own implementation for the assignment and provide a discussion about my solution. To conclude this report, I will provide a summary and a reflection on the assignment as a whole.

2 Theory of Topic

In this section I will describe the concepts and protocols used to realise my solution for this publish/subscribe protocol. An understanding in each of these areas is essential before continuing on to subsequent parts in this report, such as the implementation and the discussion, as these are the areas that this publish/subscribe protocol revolves around.

The components that I deem necessary to discuss are:

- ❖ Internet Protocol (IP)
- ❖ User Datagram Protocol
- ❖ Docker

2.1 Internet Protocol (IP)

The Internet Protocol (IP) is a protocol, or set of rules, used to enable packets to travel from network to network and arrive at the intended destination. Every device that connects to the Internet is given its own IP address. We call the pieces of data that are passed from one host to another a “packet”. Internet Protocol information such as the unique IP address is attached to each packet with the purpose of guiding the packets in the right direction. Once the packets arrive at their destination, their handling is dependent on which transport protocol is used in combination with IP. In this assignment, and in my solution, the transport protocol used is User Datagram Protocol (UDP), as discussed in the next section.

2.2 User Datagram Protocol (UDP)

User Datagram Protocol is a protocol in the Transport Layer of the OSI model. Messages sent using the user datagram protocol are called datagrams. There are several aspects of a user datagram

protocol worth noting: The UDP header is 8 bytes long, has a source and destination port number, both 16 bits long, and finally a length and checksum field, also both 16 bits long. The User Datagram Protocol is a connectionless, unreliable protocol. UDP does not feature any flow or error control, and port numbers are used to multiplex data. Although UDP does not provide any flow or error control, it is a convenient transport-layer protocol for applications that provide their own flow and error control. When speed is the priority in your protocol, and lost packets is not a big concern, then user datagram protocol may be a great option.

2.3 Communication

The basic communication and topology of my solution is the following:

- ❖ The dashboard subscribes to topics at the Broker,
- ❖ Actuators subscribe to instructions for controlling the temperature of a specific room,
- ❖ The sensors publish temperature readings for specific topics and subtopics to the Broker.
- ❖ The broker forwards these published packets to the corresponding subscribers (if the dashboard has subscribed to the published packet's topic)
- ❖ The dashboard receives the published temperature readings and sends instructions to the broker to forward the instructions to the actuator responsible for controlling the temperature in that room
- ❖ The broker forwards the dashboard's instructions to the relevant actuator
- ❖ The actuator carries out the dashboard's instructions and publishes a packet to the dashboard (through the broker) indicating that the instructions were completed as per the dashboard's request.

How I ensure the successful transmission of a packet from the source address to the destination address is discussed in the next section.

2.4 Packet Description

The composition of the packets in my solution are made up of 2 elements: the header and the payload. The first byte in the header is how any component in my solution (the dashboard, the broker, a sensor or an actuator) knows where the received packet has come from – it is used to set the destination address and to determine what happens next. There are 7 possible values for this first byte:

- 1) Dashboard Subscription
- 2) Broker
- 3) Sensor Publication
- 4) Actuator Subscription
- 5) Actuator Publication
- 6) Dashboard Publication
- 7) Acknowledgement

Once the component in question knows where the received packet has come from it can set this header value at byte position 1 to point to the next stop in this packet's IP address journey.

After analysing the header, the user input is handled to determine the topic and subtopics, taking the first input as the topic, the second as the subtopic and the remaining input as the temperature reading.

2.5 Docker

Initially, my solution provided functionality to subscribe to topics at the Broker, and the Broker forwarded published packets to the corresponding subscribers, all on my local machine. For the final submission, I have added functionality for my solution to work on any virtual machine via the use of docker and containers.



Figure 1: An image of the various containers I used in my solution.

3 Implementation

The following are the components used in my implementation:

- ❖ Node
- ❖ Acknowledgements
- ❖ Dashboard
- ❖ Broker
- ❖ Sensors (Publishers)
- ❖ Actuators (Subscribers)

3.1 Node

The Node class used in my solution was provided on Blackboard. The key aspect of the Node class is the Listener, which is an internal class in Java, extending the Thread class. The functionality of the Listener, is that it waits for incoming packets endlessly. All of my components, my Broker, Dashboard, Sensors and Actuators all extend this Node class, since they all require this functionality of waiting endlessly for incoming packets. Whenever a packet is received, the provided onReceipt method is called. The onReceipt method is an abstract method in the Node class, indicating that each class which extends the Node class, i.e., my Dashboard, Broker, Sensors and Actuators, will have their own implementation of this method.

3.2 Acknowledgements (ACKs)

Acknowledgements are included in my solution, to represent successful receipt of a packet. Whenever the dashboard, broker, sensors or actuators receive a packet, they send an acknowledgement to the sender's address, to let the sender know their packet has been received successfully. To send an acknowledgement, the recipient sets the first byte in the header to the corresponding value, i.e., 7 (see Packet Description above), and then sets the destination address as the source address of the received packet.

The dashboard, broker, sensors and actuators all have the functionality of sending an

```
public synchronized String sendACK(DatagramPacket packet, byte[] data) throws Exception {
    try {
        String content;

        byte[] buffer = new byte[data[LENGTH_POS]];
        buffer = new byte[data[LENGTH_POS]];
        System.arraycopy(data, HEADER_LENGTH, buffer, 0, buffer.length);

        content = new String(buffer);

        data = new byte[HEADER_LENGTH];
        data[TYPE_POS] = TYPE_ACK;
        data[ACKCODE_POS] = ACK_ALLOK;

        DatagramPacket response;
        response = new DatagramPacket(data, data.length);
        response.setSocketAddress(packet.getSocketAddress());
        socket.send(response);
        System.out.println("ACK sent from Broker");
        return content;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return "";
}
```

Listing 1: The method used to send an Acknowledgement. The content of the payload is returned to show the user what the sent packet's message was. "data[TYPE_POS] = TYPE_ACK" lets the future recipient know what type of packet this is (an ACK). The "response" datagram packet's destination address is set to be the received packet's address. The ACK is sent.

acknowledgement. For example, when the dashboard publishes instructions to the broker, the broker sends an ACK to the dashboard to inform the dashboard its instructions were received. When the broker forwards a temperature reading to the dashboard, the dashboard will send an acknowledgement to the Broker upon successful receipt, to let the Broker know the dashboard has received the sent packet. After publishing a packet, the publisher would wait for an acknowledgement.

3.3 Dashboard

The dashboard is an example of a type of subscriber (who also publishes!) implemented in my solution. There are 3 main events which involve the dashboard:

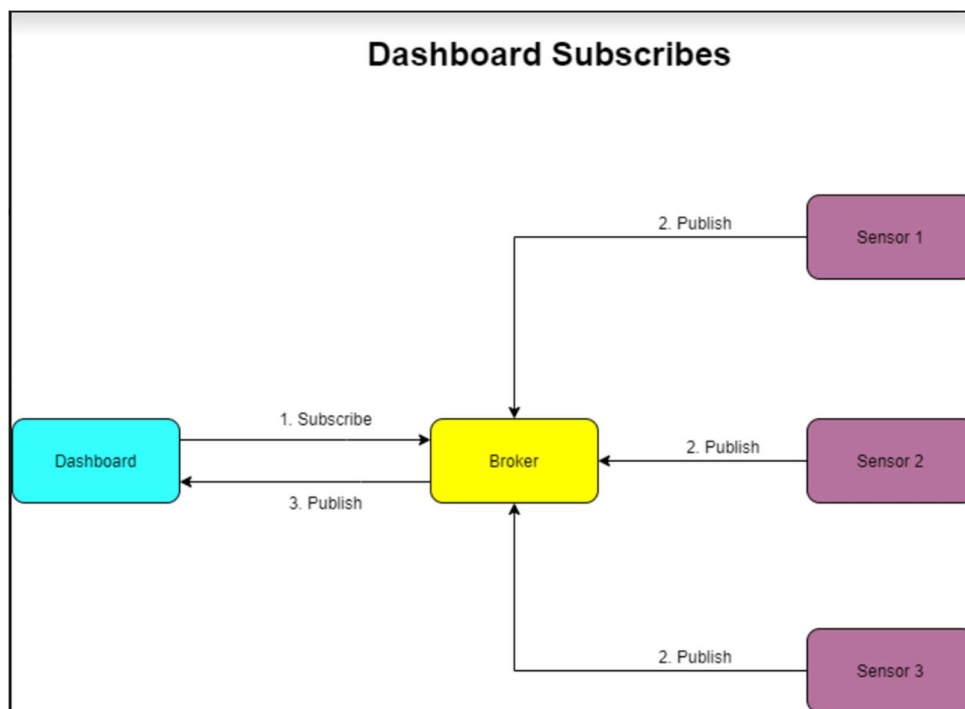
Firstly, the dashboard publishes a message to the Broker indicating which topic & subtopics it would like to subscribe to. i.e., the dashboard subscribes to topics & subtopics.

```
public synchronized void subscribeToBroker() throws Exception {
    byte[] data= null;
    byte[] buffer= null;
    DatagramPacket packet= null;
    String input;
    System.out.println("Payload: ");
    input = scanner.nextLine();
    scanner.close();
    buffer = input.getBytes();

    data = new byte[HEADER_LENGTH+buffer.length];
    data[TYPE_POS] = DASHBOARD_SUBSCRIBE;
    data[LENGTH_POS] = (byte)buffer.length;
    System.arraycopy(buffer, 0, data, HEADER_LENGTH, buffer.length);
    packet= new DatagramPacket(data, data.length);
    packet.setSocketAddress(dstAddress);
    socket.send(packet);
    System.out.println("Packet sent from Dashboard to Broker");
}
```

Listing 2: Without the dashboard subscribing, the Broker would not have access to the dashboard's IP address, hence, the dashboard subscribes. The topic and subtopics are extracted from the user input as seen in the Broker section of this report.

Secondly, the dashboard then receives packets containing information on these topics & subtopics that were published from the different sensors,



Listing 3: The dashboard subscribes at the Broker for data about a given topic. E.g., temperature readings. After subscribing, the dashboard then receives all the data packets about this topic which are published by the sensors.

Thirdly, upon receipt of a temperature reading, the dashboard would send instructions to the corresponding actuator on how to control the temperature. I have implemented 3 scenarios in my protocol:

- 1) If the temperature is greater than 30 degrees, dashboard publishes message to corresponding actuator to “lower the temperature”,
- 2) If the temperature is less than 25 degrees, dashboard publishes message to corresponding actuator to “raise the temperature”,
- 3) Otherwise, (if the temperature is not between 25 – 30 degrees), dashboard publishes message to corresponding actuator “good job”.

```
public synchronized void sendInstruction(String content) throws Exception {
    String[] contentWords = content.split(" ");
    for(String s : contentWords) {
        s.trim();
    }
    String topic = contentWords[0];
    topic+="Instructions";
    String temperatureValueString = contentWords[1];
    if(!temperatureValueString.equalsIgnoreCase("Instructions_completed_as_per_request")) {
        int temperatureValue = Integer.parseInt(temperatureValueString);

        String instruction = topic;
        if(temperatureValue>30) {
            instruction += " lower_temperature";
        }
        else if(temperatureValue<25) {
            instruction += " raise_temperature";
        }
        else {
            instruction += " good_job";
        }

        byte[] data= null;
        byte[] buffer= null;
        DatagramPacket packet= null;

        buffer = instruction.getBytes();

        data = new byte[HEADER_LENGTH+buffer.length];

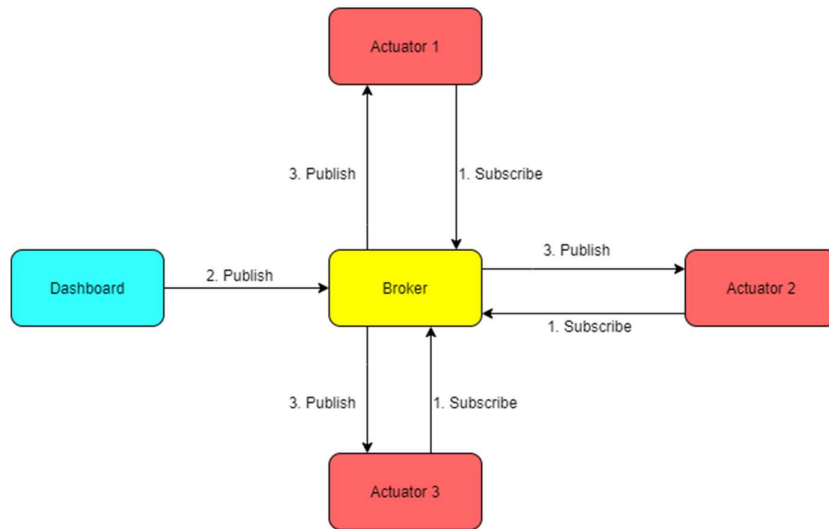
        data[TYPE_POS] = DASHBOARD_PUBLISH;

        data[LENGTH_POS] = (byte)buffer.length;

        System.arraycopy(buffer, 0, data, HEADER_LENGTH, buffer.length);
        packet= new DatagramPacket(data, data.length);
        packet.setSocketAddress(dstAddress);
        socket.send(packet);
        System.out.println("Instruction sent from Dashboard to Broker");
    }
}
```

Listing 4: The input “content” is the message received from the sensor publishing the temperature reading for the specified room. This method is only called when the dashboard receives a temperature reading for a room it has subscribed to. This code illustrates the 3 scenarios described above, before this listing.

Dashboard Publishes Instruction



Listing 5: The Actuators subscribe to data about given topics at a broker, e.g., instructions for room1. After subscribing, the actuators receive data published by the dashboard about their topic subscriptions.

3.4 Broker

The purpose of the Broker in my implementation is to act like a middle-man, as described in a publish/subscribe protocol. Any communication from one port to another goes through the Broker. How I have implemented my protocol is to use a HashMap which stores the different topics each with a list of their corresponding subscribers. Whenever the Broker receives a packet containing information on some topic &/or subtopic, the Broker then consults its HashMap to determine where the packet should be forwarded to. The main hub of activity for the Broker is in the onReceipt method.


```

public synchronized void onReceipt(DatagramPacket packet) {
    try {
        String content;
        byte[] data;
        data = packet.getData();

        SocketAddress srcAddress;

        switch(data[TYPE_POS]) {

            case DASHBOARD_SUBSCRIBE:
                System.out.println("Broker received subscription request from Dashboard");
                content = sendACK(packet, data);
                srcAddress = packet.getSocketAddress();
                checkSubscriptionExistsAndUpdate(content, srcAddress);
                break;

            case TYPE_ACK:
                System.out.println("Broker received ack");
                break;

            case ACTUATOR_SUBSCRIBE:
                System.out.println("Broker received subscription request from Actuator");
                content = sendACK(packet, data);
                srcAddress = packet.getSocketAddress();
                checkSubscriptionExistsAndUpdate(content, srcAddress);
                break;

            case SENSOR_PUBLISH:
                System.out.println("Broker received packet from Sensor");
                content = sendACK(packet, data);
                sendMessage(content);
                break;

            case DASHBOARD_PUBLISH:
                System.out.println("Broker received packet from Dashboard");
                content = sendACK(packet, data);
                sendMessage(content);
                break;

            case ACTUATOR_PUBLISH:
                System.out.println("Broker received packet from Actuator");
                content = sendACK(packet, data);
                int instructionsIndex = content.indexOf("Instructions");
                String firstHalf = content.substring(0, instructionsIndex);
                String secondHalf = content.substring(instructionsIndex+12, content.length());
                content = firstHalf+secondHalf;
                sendMessage(content);
                break;

            default:
                System.out.println("Unexpected packet: " + packet.toString());
        }
    } catch (Exception e) {if (!(e instanceof SocketException)) e.printStackTrace();}
}

```

Listing 6: The switch statement in the onReceipt method in the Broker class handles the different possible packets received and what to do next once received.

When the Broker receives a subscription request, the following code is called upon:

```

public static void checkSubscriptionExistsAndUpdate(String content, SocketAddress subscriberAddress) {
    String[] contentWords = content.split(" ");
    String topic = contentWords[0];
    ArrayList<SocketAddress> l = new ArrayList<SocketAddress>();

    // if topic already has some subscribers, then add this new subscriber
    if((l = subscriberMap.get(topic))!=null) {
        if(l.contains(subscriberAddress) == false) {
            ArrayList<SocketAddress> l2 = l;
            l2.add(subscriberAddress);
            subscriberMap.replace(topic, l, l2);
        }
    }
    // if no subscriptions to this topic, add new subscriber to this topic
    else {
        ArrayList<SocketAddress> list = new ArrayList<SocketAddress>();
        list.add(subscriberAddress);
        subscriberMap.put(topic, list);
    }
}

```

Listing 8: If the subscriber is already subscribed to the specified topic, it is not subscribed for a second time because this would result in the subscriber receiving duplicate packets. The topic is extracted from user input, and if no subscription exists to this topic for the specified subscriber, the subscription is recorded.

Alternatively, if the Broker receives a publication, the following code is called upon:

```

public synchronized void sendMessage(String contentAsString) throws Exception {

    String[] contentWords = contentAsString.split(" ");
    for(String s : contentWords) {
        s.trim();
    }
    String topic = contentWords[0];

    ArrayList<SocketAddress> l = new ArrayList<SocketAddress>();

    // if topic already has some subscribers, then add this new subscriber
    l = subscriberMap.get(topic);

    if(l!=null) {
        byte[] data= null;
        byte[] buffer= contentAsString.getBytes();
        DatagramPacket packet= null;
        data = new byte[HEADER_LENGTH+buffer.length];
        data[TYPE_POS] = BROKER;
        data[LENGTH_POS] = (byte)buffer.length;
        System.arraycopy(buffer, 0, data, HEADER_LENGTH, buffer.length);

        packet= new DatagramPacket(data, data.length);
        System.out.println("Forwarding packet from Broker...");

        for(SocketAddress dstAddress : l) {
            packet.setSocketAddress(dstAddress);
            socket.send(packet);
            System.out.println("Packet sent");
        }
    }
}

```

Listing 7: Extracting the topic from user input, and searching for the subscribers to this topic in its HashMap, the broker forwards the published packet to the subscribers to this topic.

3.5 Sensor (Publishers)

In my implementation I have instantiated 3 sensors, with the idea being each sensor would publish a temperature reading for a certain room. This design helps illustrate the functionality of subscribing to subtopics. For simplicity, sensor1 reports temperature readings for room1, sensor2 reports temperature readings for room2 and sensor3 reports temperature readings for room3. The sensors

are the components of my solution which will publish the data about certain topics &/or subtopics that my dashboard may/may not be subscribed to. The code for the sensor publishing a message is

3.6 Actuator (Subscribers)

Similar to my sensors, my implementation involves 3 actuators. Just like my sensors, each actuator is assigned to a specific room. Actuator1 is responsible for controlling the temperature in room1, actuator2 is responsible for controlling the temperature in room2 and actuator3 is responsible for controlling the temperature in room3.

The main functionality of each actuator is to continuously wait for incoming instructions from the dashboard, about how the dashboard wants to/does not want to change the state of the temperature in the corresponding room. Once instructions have been received the actuator executes these instructions (to receive the instructions each actuator must subscribe to the topic; the code for this is very similar to the Dashboard's subscription request seen in listing 2. For the sake of not repeating myself and not making this report unnecessarily long with repeated code I have not included this code again!).

After receiving an instruction from the dashboard, the dashboard then publishes a message of its own back to the dashboard, indicating that the instructions were carried out. Why I included this in my design is the idea that the dashboard can rest assured the temperature in a given room will be controlled adequately.

```
public synchronized void executeInstruction(String content) throws Exception {
    String[] contentWords = content.split(" ");
    for(String s : contentWords) {
        s.trim();
    }
    String topic = contentWords[0];
    String instruction = contentWords[1];

    if(instruction.equalsIgnoreCase("good_job")) {
        System.out.println("Continuing as normal");
    }
    else if(instruction.equalsIgnoreCase("lower_temperature")) {
        System.out.println("Lowering temperature");
    }
    else if(instruction.equalsIgnoreCase("raise_temperature")) {
        System.out.println("Raising temperature");
    }
}

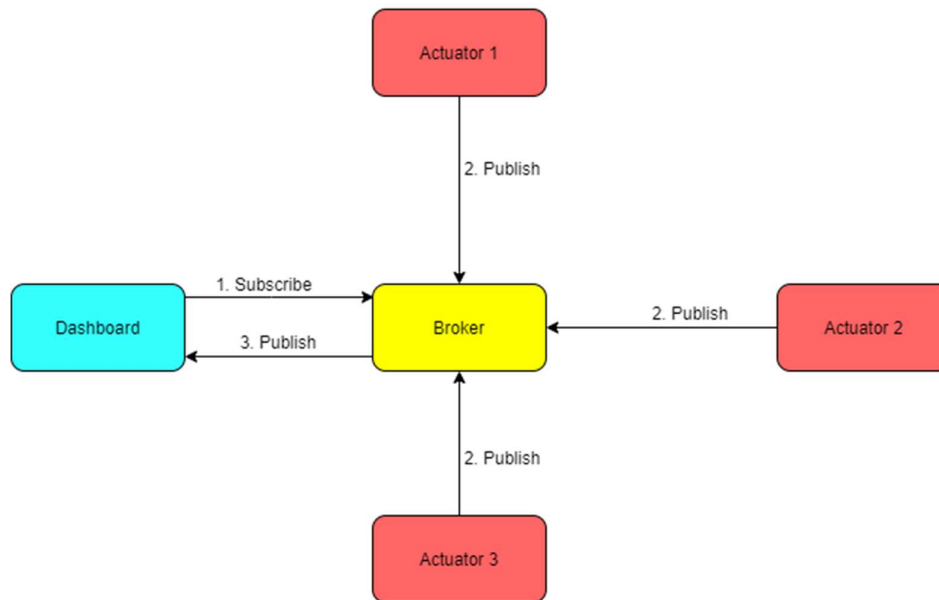
byte[] data= null;
byte[] buffer= null;
DatagramPacket packet= null;
String reply = topic + " Instructions_completed_as_per_request";
buffer = reply.getBytes();
data = new byte[HEADER_LENGTH+buffer.length];
data[TYPE_POS] = ACTUATOR_PUBLISH; // To show ACTUATOR is sending message
data[LENGTH_POS] = (byte)buffer.length;

System.arraycopy(buffer, 0, data, HEADER_LENGTH, buffer.length);

packet= new DatagramPacket(data, data.length);
packet.setSocketAddress(dstAddress); // set socketAddress to Broker address
socket.send(packet);
System.out.println("Packet sent from Actuator to Broker");
}
```

Listing 9: Function for actuator to execute the instruction received from the Dashboard. Once the instruction has been received, the actuator publishes a message of its own to the dashboard, simply to indicate that the instructions have been carried out as requested, just to put the dashboard's mind at ease.

Actuator(s) Publish Instruction Completed



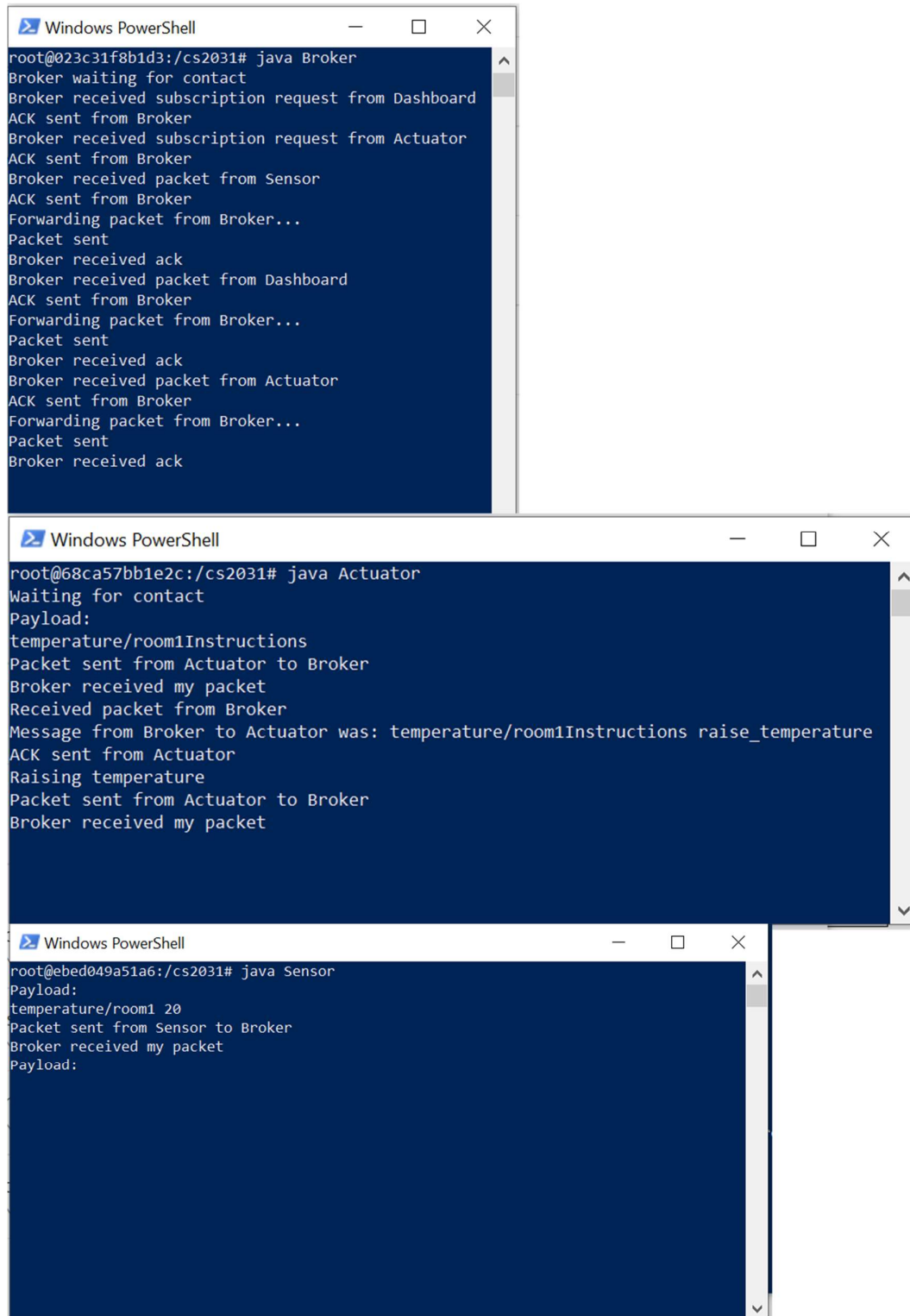
Listing 10: The actuator publishes a message to the dashboard informing the dashboard the instructions were carried out.

4 Discussion

4.1 My Design

Since a picture is worth a thousand words, I have attached some demonstrative snapshots of the communication in my design between my components below:

```
Windows PowerShell
root@2cef979de0c9:/cs2031# java Dashboard
Waiting for contact
Payload:
temperature/room1
Packet sent from Dashboard to Broker
ACK received from Broker
Received packet from Broker
Message from Broker to Dashboard was: temperature/room1 20
ACK sent from Dashboard
Instruction sent from Dashboard to Broker
ACK received from Broker
Received packet from Broker
Message from Broker to Dashboard was: temperature/room1 Instructions_completed_as_per_request
ACK sent from Dashboard
```



```
Windows PowerShell
root@023c31f8b1d3:/cs2031# java Broker
Broker waiting for contact
Broker received subscription request from Dashboard
ACK sent from Broker
Broker received subscription request from Actuator
ACK sent from Broker
Broker received packet from Sensor
ACK sent from Broker
Forwarding packet from Broker...
Packet sent
Broker received ack
Broker received packet from Dashboard
ACK sent from Broker
Forwarding packet from Broker...
Packet sent
Broker received ack
Broker received packet from Actuator
ACK sent from Broker
Forwarding packet from Broker...
Packet sent
Broker received ack

Windows PowerShell
root@68ca57bb1e2c:/cs2031# java Actuator
Waiting for contact
Payload:
temperature/room1Instructions
Packet sent from Actuator to Broker
Broker received my packet
Received packet from Broker
Message from Broker to Actuator was: temperature/room1Instructions raise_temperature
ACK sent from Actuator
Raising temperature
Packet sent from Actuator to Broker
Broker received my packet

Windows PowerShell
root@ebcd049a51a6:/cs2031# java Sensor
Payload:
temperature/room1 20
Packet sent from Sensor to Broker
Broker received my packet
Payload:
```


	Time	Source	Destination	Protocol	Length	Info
4	0.000717	172.20.0.2	172.20.0.3	UDP	46	50001 → 50002 Len=2
5	10.427818	172.20.0.5	172.20.0.2	UDP	75	50008 → 50001 Len=31
6	10.427855	172.20.0.5	172.20.0.2	UDP	75	50008 → 50001 Len=31
7	10.428263	172.20.0.2	172.20.0.5	UDP	46	50001 → 50008 Len=2
8	10.428273	172.20.0.2	172.20.0.5	UDP	46	50001 → 50008 Len=2
9	18.845462	172.20.0.4	172.20.0.2	UDP	66	50003 → 50001 Len=22
10	18.845522	172.20.0.4	172.20.0.2	UDP	66	50003 → 50001 Len=22
11	18.845926	172.20.0.2	172.20.0.4	UDP	46	50001 → 50003 Len=2
12	18.845940	172.20.0.2	172.20.0.4	UDP	46	50001 → 50003 Len=2
13	18.847272	172.20.0.2	172.20.0.3	UDP	66	50001 → 50002 Len=22
14	18.847286	172.20.0.2	172.20.0.3	UDP	66	50001 → 50002 Len=22
15	18.848208	172.20.0.3	172.20.0.2	UDP	46	50002 → 50001 Len=2
16	18.848223	172.20.0.3	172.20.0.2	UDP	46	50002 → 50001 Len=2
17	18.849501	172.20.0.3	172.20.0.2	UDP	93	50002 → 50001 Len=49
18	18.849519	172.20.0.3	172.20.0.2	UDP	93	50002 → 50001 Len=49
19	18.849930	172.20.0.2	172.20.0.3	UDP	46	50001 → 50002 Len=2
20	18.849947	172.20.0.2	172.20.0.3	UDP	46	50001 → 50002 Len=2
21	18.850310	172.20.0.2	172.20.0.5	UDP	93	50001 → 50008 Len=49
22	18.850331	172.20.0.2	172.20.0.5	UDP	93	50001 → 50008 Len=49
23	18.851301	172.20.0.5	172.20.0.2	UDP	46	50008 → 50001 Len=2
24	18.851316	172.20.0.5	172.20.0.2	UDP	46	50008 → 50001 Len=2
25	18.852583	172.20.0.5	172.20.0.2	UDP	113	50008 → 50001 Len=69
26	18.852596	172.20.0.5	172.20.0.2	UDP	113	50008 → 50001 Len=69

Listing 12: A sample of the traffic captures of my solution captured by Wireshark while running my solution on Docker.

Below I will discuss some aspects of my design decisions, including some advantages and disadvantages, and why I chose the design decisions I chose.

4.2 Advantages of my Design

One design decision I have chosen, and feel is an advantage, is that the subscriber (the dashboard) is not restricted to subscribing to a finite list of topics and subtopics. Admittedly this has caused some possible errors depending on user input, but I wanted my user to have the freedom to subscribe to any topic. For example, my example illustrates a dashboard, sensors, and actuators communicating about temperature levels, but what if I want to use my solution for communicating data about carbon dioxide levels, or any other topic that comes to mind? I could have easily designed my solution to only handle hard-coded topic names, such as temperature levels, but then I could not communicate carbon dioxide levels. One could argue to just hard code carbon dioxide levels as well, but the more topics you want to cover, the more tedious and inefficient this solution would be in my opinion. This is why I have chosen the design decision of assigning the topic to the user's input.

Another advantage of my design (again in my opinion!) is that the actuator sends a confirmation message indicating that it received the dashboard's instructions. Technically, an ACK from the Broker to the dashboard and an ACK from the actuator to the Broker is sufficient for this, but I wanted to add an element of "extra communication" to this assignment.

For this same reason, I also made the design decision of adding the subscriber addresses of a given topic to my HashMap by extracting the address from the received packet's source address. I could have hard-coded some topic subscription-address pairs to my broker's HashMap and avoided the initial subscription request made by the dashboard and the actuators, but I prefer the idea of having the ability to cover a wider range of topics.

4.3 Disadvantages of my Design

My design is dependent on specific user input and therefore is prone to errors. Why I did not fix these errors before my final submission is simply due to time constraints, as I prioritised getting topic functionality working, as well as Docker, before moving on to the error checking.

One aspect of my design that I would include if I was to re-do this assignment would be to implement a time-out error or perhaps the stop and wait protocol. The reason why this is not included in my overall solution is again down to time constraints.

There were several functionalities I wanted to include in my design and was looking forward to implementing these, but, again, I just ran out of time. Such functionalities were the ability to unsubscribe to a topic, the creation of another subscriber in addition to the dashboard, perhaps a floor/office manager who wanted to subscribe to different topics.

Although there are additional functionalities I could have included to improve my final solution, I am still very happy with my final solution considering this was my first attempt at a publish/subscribe protocol!

5 Summary

This report has described my attempt at a solution to address a publish/subscribe protocol. The theory of the topic was discussed and my implementation for my solution was covered. The description of the implementation in this document highlights the essential components of my solution and demonstrates the execution of the solution in an example topology, the example being:

- ❖ a dashboard subscribes to temperature readings in a certain room,
- ❖ receives the readings through sensor reading reports,
- ❖ the dashboard sends instructions to actuators based on the readings,
- ❖ and finally the actuators let the dashboard know the temperature is under control!

6 Reflection

I found the task of this assignment to be very helpful and effective in terms of furthering my learning and understanding of the topics and content taught in the lectures. I found the practical task of designing our own solution to a publish/subscribe protocol encouraged me to engage more with the lecture material and that the assignment as a whole was extremely beneficial in terms of learning. I feel that the freedom to choose how our protocol works and is designed allows us to explore more possibilities and avenues and is a refreshing way for us to approach what we have learnt in our lectures. I had a lot of fun with this assignment and am looking forward to the next one!