Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33031 Computer Networks
Assignment #1: Protocols

Sean Gaffney, Std# 19304695

October 27, 2021

1 Introduction

This assignment tasked us with creating a simple Publish/Subscribe protocol that would allow multiple simulated devices to exchange small amounts of data. This protocol should in theory be similar to MQTT-SN(Message Query Telemetry Transport for Sensor Networks), a UDP (User Datagram Protocol) implementation for low power IOT devices exchanging very basic sensor data. The basic idea of a publish/subscribe protocol is that a device may subscribe to a given topic via a central broker. Other devices may publish data to that topic that the broker will then pass on to all subscribers. My implementation is written in Java and uses Docker to create an example network with multiple devices that can exchange data based on given topics. For this implementation I used Java, and extended the sample code given in the Docker folder on blackboard. In this example, the various nodes are exchanging imaginary temperature and humidity data to make basic decisions such as opening and closing a window.

2 Theory of Topic

This section discusses the major concepts that are core to understanding how this protocol works.

2.1 Internet Protocol

Internet Protocol (IP) is a protocol in the Network Layer of the Open Systems Interconnection (OSI) model. Two versions of IP are currently used on the Internet, IP version 4 (IPv4) and IP version 6 (IPv6). In IPv4 the IP address is 4 bytes long, whereas in IPv6 it is 16 bytes long. In both versions, the IP header contains the IP address of the source host and of the destination host. Routers use these IP addresses to forward packets to the correct destination. In Java, an IP Address is represented by an InetAddress object, which may use either IPv4 or IPv6.

2.2 User Datagram Protocol

User Datagram Protocol is a protocol in the Transport Layer of the OSI model that builds on top of IP. A UDP header is 8 bytes long. It has a 16-bit source port number, a 16-bit destination port number, a 16-bit total length field and a 16-bit checksum field. Port numbers are used to identify a specific process to which a message is forwarded when it arrives at its destination. The checksum field is used for error control but it is optional. Because of its short length and small number of fields, UDP is a connectionless and unreliable protocol. However its lack of flow control and optional error control can be beneficial in applications that provide these themselves.

3 Implementation

This section describes the implementation of the publish/subscribe protocol including the makeup of the packets, as well as the topology of the network and a description of the different types of Nodes. This project is dockerized, meaning each Node is running in its own Docker container. Nodes interact over a Docker virtual network, titled “cs2031”. This means that nodes can refer to each other with the name of the docker container instead of a traditional IP address. The name of the docker container that a Node is running in must correspond with its address listed in the Node class. The broker is running on a Ubuntu container that is also running Wireshark, while all other Nodes can be run in an openJDK container.

3.1 Packet Description

Packets using this protocol have four components, the UDP header, the packet type, packet topic, and depending on the type of packet, a message. The type, topic and message are all contained in the UDP buffer. The type is defined in the first byte, the topic in the second byte, and the remainder of the buffer, if any, is assumed to be a String.
There are four packet types, ACKPACKET(acknowledgment packets), SUBPACKET(subscription packets), and MESSAGE(message/data packets). Each packet type Extends the abstract PacketContent class. PacketContent defines constants for the packet types and functions for creating and receiving Datagram packets. Within the class definition for each packet type are the specifics for adding relevant information. Not all packet types require a topic or message, and will leave those fields as 0. ACKPACKET has no topic, and a given message. SUBPACKET only has the topic it is requesting and no message. Note that because this packet is only two bytes and no text, it may not be clear what it is when picked up by Wireshark. MESSAGE has both the topic it is publishing to, and a given message.

3.2 Node

Node is an abstract class that is extended by Broker, Subscriber and Client. It is mostly unchanged from the given version from Blackboard. The main functionality that we were provided was an internal class called Listener, which extends java.lang.Thread. The listener waits endlessly for incoming packets. When a packet is received the onReceipt method is called. onReceipt is an abstract method so each subclass of Node must handle this in its own way. Functionality and constants needed by all types of Nodes was all added.

3.3 Broker

The Broker extends Node, its purpose is to route traffic between other Nodes and to maintain the list of topics and what nodes are subscribed to each. Each topic has an ArrayList of SocketAddress that allows the broker to track subscriptions. The broker runs indefinitely and listens on its given port. When a packet is received it extracts the topic and type from the packet to decide how it should be routed using switch statements. If an Acknowledgement packet is received, confirmation is printed to the terminal. If a Message packet is received, the broker will forward it to all known subscribers. If a Subscription packet is received, it will extract the sender's SocketAddress and the topic byte, and add the address to the corresponding topic ArrayList.

```
packetCount++;
          System.out.println("Received packet: " + packetCount);


          DatagramPacket response;
          response = new AckPacketContent("OK - Received this").toDatagramPacket();
          response.setSocketAddress(packet.getSocketAddress());
          socket.send(response);


          PacketContent content= PacketContent.fromDatagramPacket(packet);
          int packetType = content.getType();
          int packetTopic = content.getTopic();
          switch(packetType) {
```

```
                case PacketContent.ACKPACKET:
                    System.out.print("Received Ack packet");
                    break;
                case PacketContent.MESSAGE:
                    System.out.println("Received Message packet, forwarding to
subscribers");
                    DatagramPacket copyPacket = packet;
                    switch(packetTopic) {
                        case PacketContent.TEMP:
                            System.out.println("Forwarded to TEMP");
                            for(InetSocketAddress i : temperature) {
                                copyPacket.setSocketAddress(i);
                                socket.send(copyPacket);
                            }
                            break;
                        case PacketContent.HUMIDITY:
                            System.out.println("Forwarded to HUMIDITY");
                            for(InetSocketAddress i : humidity) {
                                copyPacket.setSocketAddress(i);
                                socket.send(copyPacket);
                            }
                            break;
                        case PacketContent.NOTOP:
                        default:
                            System.err.println("Error: Unexpected packet received
topic:" + packetTopic);
                            break;
                    }
                    break;
                case PacketContent.SUBPACKET:
                    System.out.println("Received  Sub packet, adding sender to
subscribers");
                    if(packetTopic == PacketContent.TEMP) {
                        temperature.add((InetSocketAddress)
packet.getSocketAddress());
                        System.out.print(""+temperature.size());
                    } else if (packetTopic == PacketContent.HUMIDITY) {
                        temperature.add((InetSocketAddress)
packet.getSocketAddress());
                        System.out.print(""+humidity.size());
                    }
                    else {
                        System.err.println("Error: Unexpected packet received topic:"
```

```
+ packetTopic);
                    }
                    break;
                default:
                    System.err.println("Error: Unexpected packet received (invalid
type)");
                    break;
            }
```

Listing 1: The logic used by the Broker in onReceipt().

3.4 Client

Client extends Node, its purpose is to generate example data about either temperature or humidity. Client only runs once, and does not wait to listen for further packets after receiving an acknowledgment. It generates random values, one of which is used to determine the topic, either temperature or humidity, and the second, to be the value sent. It creates a message packet with the given topic and message and then closes once receiving confirmation.

```
RRandom rand = new Random();
        int topic = rand.nextInt(10); //randomly picks a topic to publish to
        int value = rand.nextInt(100);

        DatagramPacket packet;
        if(topic > 5) {
            packet = new Message(PacketContent.TEMP, "Temperature:" +
value).toDatagramPacket();
            System.out.println("Temperature:" + value);

        }
        else {
            packet = new Message(PacketContent.HUMIDITY,"Humidity:" +
value).toDatagramPacket();
            System.out.println("Humidity:" + value);
        }
        packet.setSocketAddress(dstAddress);
        socket.send(packet);
        System.out.println("Packet sent");
        this.wait();
```

Listing 2: The start() function, where the Client generates data and publishes it to the give tomic

3.5 Subscriber

Subscriber extends Node, its purpose is to receive data and simulate an actuator. In this case it

decides to either open or close a window based on the current temperature. When started the subscriber sends a subscription request to the Broker for the TEMP topic. It then waits indefinitely for an update. If or when it receives a message, it extracts the temperature data and decides if the window should be opened or closed, it provides an update to the command line.

```
        public synchronized void onReceipt(DatagramPacket packet) {
    try {
        System.out.println("Packet was Recieved");

        PacketContent content= PacketContent.fromDatagramPacket(packet);

        if (content.getType()==PacketContent.MESSAGE) {
            System.out.println(content.toString());
            String[] arrOfStr = content.toString().split(":", 2);
            int tempVal = Integer.parseInt(arrOfStr[1]);
            if(tempVal > 50 && windowOpen == false) {
                windowOpen = true;
                System.out.println("Opened the window");
            }
            else if (tempVal < 50 && windowOpen == true) {
                windowOpen = false;
                System.out.println("Closed the window");
            }
            else {
                System.out.println("Did not move the window");
            }

        }
        //this.notify();
    }
    catch(Exception e) {e.printStackTrace();}

}

public synchronized void start() throws Exception {

    DatagramPacket request;
    request= new SubContent(PacketContent.TEMP).toDatagramPacket();
    request.setSocketAddress(dstAddress);
    socket.send(request);
    System.out.println("" + new SubContent(PacketContent.TEMP).getTopic());
    System.out.println("Waiting for contact");
    this.wait();
```

```
    }
```

Listing 3: The onReceipt() and Start() functions of the Subscribe. In onReceipt() it parses a given message and actuates the “window” accordingly. In Start() it sends a subscribe request for temperature to the Broker in order to get the necessary temperature data.

4 Discussion

The strengths of this implementation is the design of the packets themselves. They are very lightweight, the only addition in length from the given implementation is a single byte for the topic. This means that the implementation is still very efficient. Once the network and Nodes are initialized, communication between nodes is essentially instantaneous. There is no reason to think this protocol would not work on a larger network. In the given implementation you could add as many Clients and Subscribers as necessary, as long as other Nodes have the Brokers ip address, which is hardcoded in, they should be able to publish and subscribe to topics.

This project is dockerized, meaning each Node is running in its own Docker container. Nodes interact over a Docker virtual network, titled “cs2031”. This means that nodes can refer to each other with the name of the docker container instead of a traditional IP address. However these container names could easily be replaced with IP addresses if the protocol was implemented elsewhere.

4.1 Limitations

There are a few limitations of my implementation, that mainly came down to time constraints. These mainly have to do with aspects of the various Nodes that are hardcoded to decrease the complexity of their interactions. For there are only two topics, temperature and humidity. Ideally this would be scalable, with the option to add or remove topics. However this would have greatly increased complexity. Another limitation is that all interaction with the Nodes is done via the command line, as I ran into technical difficulties with x11 that I eventually decided were not worth solving as they also took up a large portion of my time. Finally I planned on implementing user interaction through a Server node that allowed the user to subscribe to topics or publish a string to a given topic. However I had to cut this as I ran out of time and I had already demonstrated basic functionality of the protocol.

5 Summary

For this assignment I implemented a rudimentary publish/subscribe protocol in Java. This solution borrows heavily from the example code found in Blackboard, however every aspect has been modified to allow for my more complex solution. I created an example Network using multiple simulated devices on a Docker network. This included a Broker that managed the flow of packets, a Client to generate data, and a subscriber to receive data. These devices communicated with each other using my custom protocol that allowed for different types of packets and to specify topics that they be sent to, as well as a data payload. Traffic was captured at the Broker by Wireshark, the submitted pcap file shows both publish, subscriber, and acknowledgement packets.

6 Reflection

I wish I could say that I enjoyed this assignment as I find networking very interesting and as I

made my way through it I had many “aha” moments and I learned about network protocols, Docker, and Wireshark. However the structure of the assignment and the module made the entire process feel unnecessarily complicated and frustrating. I never felt as if I had enough guidance on where to go with the assignment. While I also find the content of the lectures engaging, it was mostly theory that seemed very disconnected from the more practical application in the assignment. I wasted large amounts of time getting the technology, Docker and Wireshark, working which seriously hindered my ability to focus on the protocol itself. For example for Part 1 of the assignment, running Docker as its own container listening in on the network. However this stopped working consistently as I increased the complexity of the Network for Part 2. I spent hours trying to implement a solution right up until the deadline but came up short. So despite having made huge progress on my protocol, I did not have a pcap to submit. Since then I was able to remedy this by getting tShark2 working on the same container as the broker. Overall, I felt that time constraints, and lack of resources limited the scope of my protocol. My hope is that having sorted out these details in this first assignment, I will be able to better focus on the core of the second assignment without this module taking up all of my time.