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

CS2031 Telecommunication II
Assignment #1: Publish/Subscribe Protocol

Ciarán Coady, Std# 17326951

December 31, 2018

Contents

1 Introduction

For this assignment we were tasked with designing a publish/subscribe protocol and in the process of developing our solution get to know about sockets, datagram packets, threads and protocol design. I also wished to gain experience in report writing and improve my ability to explain the work I have completed.

2 Theory of Topic

In the following subsections I will explain what a publish/subscribe protocol is, Sockets & Ports, Datagram Packets and Threading. Along with the explanations I will mention some of the reasons why these items are useful for this assignment.

2.1 Publish/Subscribe Protocol

Figure 1 shows a simplified topology for a publish/subscribe protocol.

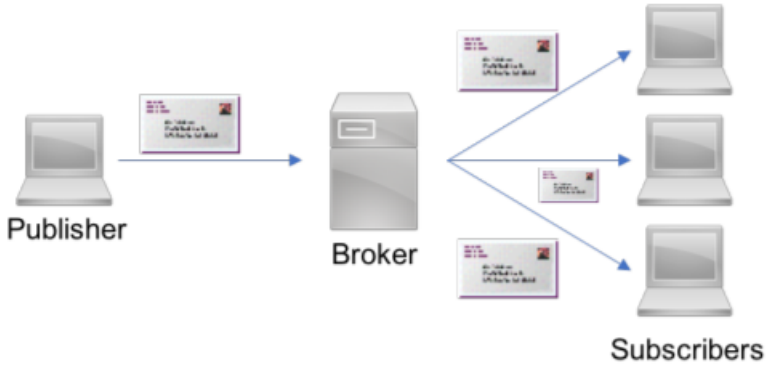

Figure 1: Shows a simple topology for a Publish/Subscribe System

A Publish/Subcribe Protocol consists of three parties: Publisher, Broker and Subscriber.

Publisher: Can create topics that can be subscribed to and publish messages for these topics.

Broker: Manages subscriptions to and unsubscriptions from topics. The broker also receives publications from the publisher for a given topic and then forwards these publications to any subscriber subscribed to that topic.

Subscriber: Can subscribe to a topic, unsubscribe from a topic and receive relevant publications from the broker relating to any subscriptions.

To implement a system of this nature, I first needed to research about sockets, datagram packets and threading.

2.2 Sockets & Ports

A socket is an endpoint for a connection, much like the name suggests. In this assignment each publisher, broker and subscriber has its own socket and port number. Then using these unique addresses, we can route traffic to the correct destination. This allows each node to send its packets to the desired recipient.

2.3 Datagram Packets

In java datagram packets are used to implement a connection-less packet delivery service. Each packet is routed from one machine to another based solely on information contained within that packet. Multiple packets sent might be routed differently and might arrive in any order at the destination; as a result, it was imperative to include some sort of packet ordering system in my implementation. Packet delivery is not guaranteed with datagram packets, resulting in the need for an acknowledgment system.
For the purposes of this assignment we must implement our own flow control using the data contained in each packet. Starting with the design of MQTT(Message Queuing Telemetry Transport), I came up with my own packet design which handles flow control and packet ordering. I also implemented some of the features in MQTT that I found to be helpful for my use case.

2.4 Threading

In the normal execution of a program, variables are assigned, methods are called and the instruction pointer moves onward and a stack is maintained. With a single program it is easy to have complete control of the entire system as no resources need to be shared.
If no measures were taken and it was attempted to execute multiple programs at once, problems would occur as both programs will be trying to access and modify some of the same resources at the same time; thus interfering with each other. A solution to this would be giving each program its own separate address space and allocating it registers. This adds complexity as now each process must be given time to execute with its registers and program counter being saved and restored when execution starts or ends. A method of scheduling each process must also be implemented. All these extra steps add overhead and in situations where latency is not tolerated (the financial sector, online gaming) this is not a viable solution.
This is where a thread comes in. Threads are lightweight processes, they share the same address spaces and introduce less overhead when switching from thread to thread, when compared to switching between processes. Threads become extremely useful when there are tasks that we want to perform concurrently.
For example, if we had a web server that only had a single thread of execution, then it would only be capable of handling one connection at a time. As you can imagine if multiple people want to connect to the same server at the same time, this will lead to problems. However, by having multiple threads this server could deal with multiple clients simultaneously.
For this project we will be using the java threads class. This will allow us to not only handle input from the user but also monitor traffic to the nodes socket.
Note* when using threads we must be careful when accessing global variables to ensure that threads do not interfere with each other due to the non-deterministic execution of the threads.

3 Implementation

This section presents the implementation details of the Publish/Subscribe system. Here I discuss the process involved in developing communications between nodes, such as the publisher and the broker. Details of packet design along with flow control are all explained, along with the code used to do so.

3.1 Node

Early development of the code started with the supplied template project, consisting of a server and a client which could send packets to each other. Both the server and client were sub-classes of the node class, which contained all the necessary code to initialize a socket and thread to listen for any incoming packets on the nodes port. Due to the fact that this code would be required for the publisher, broker and subscriber the node became the location where much of the back-end code is located. In the current implementation the node contains all of the shared code such as packet handling, as it was simplistic and easy to maintain in this manner.

The node contains a listener class which was supplied in the sample code.
The listener is an extension of the thread class that is included in java. This allows for any node to have two threads of execution; one is the mainline of the node and the other being the listener thread. Once the listener is started using the run() method, it will endlessly loop trying to receive packets, this is achieved with a while(true) loop. If a packet is received the listener will call the onReceipt() method, passing in the packet that has just been received.

The createPacketData() method takes in all of the relevant packet information and returns a byte array in the correct layout.

```
protected byte[] createPacketData(int type, int packetNumber,
        int sequenceNumber, int topicNumber, byte[] message) {
                byte[] data = new byte[PACKETSIZE];
                data[0] = (byte) type; // Set type
                data[1] = (byte) packetNumber; //Set packet number
                data[2] = (byte) sequenceNumber; //Set sequence number
                data[3] = (byte) topicNumber; // Set topic number
                for (int i = 0; i < message.length; i++) {
                        data[i + 4] = message[i]; // input message content
                }
                return data;
        }
```

Listing 1: The code above shows the way in which a packet is made

The createPackets() method takes in the type of packet that is being sent, the topic number it relates to, the message to be sent and the destination address of the packet. Then depending on the size of the message that is to be sent and the max size of a packet; it is determined how many packets is required to send the message. The contents of each packet is then populated and the correct sequence number allocated. The sequence numbers are used on the receiving end to ensure that message order is preserved. When each packet is created, it gets added to an array of datagram packets. The method then returns this array of DatagramPackets.

```
protected DatagramPacket[] createPackets(int type, int topicNumber,
                                String message, InetSocketAddress dstAddress) {
```

Listing 2: This is the function header for createPackets() method

```
 int maxMessageSize = PACKETSIZE - Constants.SIZE_OF_HEADER;
```

```
byte[] tmpArray = message.getBytes()
byte[] messageArray = new byte[tmpArray.length + 1];
for (int i = 0; i < tmpArray.length; i++) {
        messageArray[i] = tmpArray[i];
}
messageArray[tmpArray.length] = 0;
```

Listing 3: This code takes the message string and ensures that it is null terminated, this is helpful for packet handling on the receiving side of the communication

```
int numberOfPackets = 0;
for (int messageLength = messageArray.length;
                messageLength > 0; messageLength -= maxMessageSize) {
        numberOfPackets++;
}
int sequenceNumber = numberOfPackets-1;
DatagramPacket[] packets = new DatagramPacket[numberOfPackets];
int offset = 0;
for (int packetNumber = 0; packetNumber < numberOfPackets; packetNumber++) {
        byte[] dividedMessage;
        if (messageArray.length - offset > maxMessageSize) {
                dividedMessage = new byte[maxMessageSize];
        } else {
                dividedMessage = new byte[messageArray.length - offset];
        }
        for (int j = offset; j - offset < dividedMessage.length; j++) {
                dividedMessage[j - offset] = (byte) messageArray[j];
        }
        byte[] data = createPacketData(type, packetNumber,
                        sequenceNumber, topicNumber, dividedMessage);
        DatagramPacket packet = new DatagramPacket(data, data.length,

        packets[packetNumber] = packet;
        offset += maxMessageSize;
}
 return packets;
}
```

Listing 4: A for-loop is used to calculate how many packets are required to send the message. After this a second for-loop is used to split the message array up into separate packets, only if the message was too long for a single packet

The code in listing 5 shows how it is determined if a packet is the last packet in a transmission.
Due to the fact that we null terminated the message before it was put into a packet it means that there will always be a 0 byte in the last position in the last packet. So by checking the last byte of message we can determine if it is the last packet of a given transmission. It could also determined if it is the last packet of a transmission using the sequence numbers, but this method was implemented before sequence numbers were added.

```
protected boolean lastPacket(DatagramPacket packet) {
                byte[] data = packet.getData();
                if (data[data.length - 1] == 0) {
                        return true;
                } else {
                        return false;
                }
```

```
        }
```

Listing 5: Determines if a packet is the last packet of a transmission

The code in listing 6 removes the message portion of a packet and returns it as a string.
A for-loop starting at the index after the last byte of the header is used to extract all the bytes of the message from the packet. This new byte array is then converted to a string using a for-loop.

```
protected String processMessageFromPacket(DatagramPacket packet) {
  byte[] data = packet.getData();
  char[] stringInChars = new char[data.length];
  for (int index = Constants.SIZE_OF_HEADER; index < data.length &&
                                                  data[index] != 0; index++) {
        stringInChars[index - Constants.SIZE_OF_HEADER] = (char) data[index];
  }
  String topic = "";
  for (int index = 0; index < stringInChars.length &&
                                  stringInChars[index] != 0; index++) {
          topic = topic + stringInChars[index];
  }
  return topic;
}
```

Listing 6: Removes the message portion of a packet and returns it as a string.

The code in listing 7 takes a packet and returns it in string format, this was useful for debugging and determining if packets were being created correctly and also they were being transmitted and received properly.

```
protected String printPacketContent(DatagramPacket packet) {
  byte[] data = packet.getData();
  String packetContent = ("Type: " + data[0] + " PacketNo: "
        + data[1] + " SequenceNo: " + data[2] + " TopicNo: " + data[3] + "\n"
                        + "Message: " + processMessageFromPacket(packet));
  return packetContent;
}
```

Listing 7: String representation of a packet

The code in listing 8 shows how packets are being sent. In the current implementation it takes an array of datagram packets and sends each one using a for-loop, there is no re-transmission or any form of error checking in this implementation.

```
protected synchronized void sendPackets(DatagramPacket[] packets)
                                                  throws InterruptedException {
                for (int index = 0; index < packets.length; index++) {
                        try {
                                socket.send(packets[index]);
                                this.wait();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }
```

Listing 8: sendPackets() Method

3.2 Publisher

Starting with the project specification I compiled a list of basic features that the publisher must implement as stated above 1.
The Code in listing 9 shows the constructor of the publisher, this sets up its socket address and port number which is based on a constant. It also allocates memory for the hashmap that is used to store the topic numbers.

```
Publisher(Terminal terminal) {
  try {
          this.terminal = terminal;
          dstAddress = new InetSocketAddress(Constants.DEFAULT_DST_NODE,
                                                  Constants.TEST_BKR_PORT);
          socket = new DatagramSocket(Constants.TEST_PUB_PORT);
          listener.go();
  } catch (java.lang.Exception e) {
          e.printStackTrace();
  }
  topicNumbers = new HashMap<Integer, String>();
}
```

Listing 9: Constructor of the Publisher

The code in listing 10 shows the addTopicToList() method which takes a new topicNumber (which is supplied by the broker) and the topic name. It adds these to the topicNumbers hashmap, allowing for quick retrieval of topics.

```
private void addTopicToList(int topicNumber, String topicName) {
                topicNumbers.put(topicNumber, topicName);
        }
```

Listing 10: Adding a new topic to the list

The code in listing 11 shows the checkTopic() method, this is called when the publisher wants to make a new topic. The user is prompted to enter the topic name, this input is then bundled into a packet and sent to the broker. The broker maintains a list of topics and their corresponding topic numbers. If the topic doesn't exist the broker will create it and return the topic number. If the topic already exists the broker will return the topic number that corresponds to that topic.

```
private void checkTopic() throws InterruptedException {
  String topic = terminal.readString("Please enter a topic to create: ");
  terminal.println("Sending packet...");
  DatagramPacket[] packets = createPackets(Constants.CREATION,
                                  0, topic, dstAddress);
  sendPackets(packets);
  terminal.println("Packet sent");
}
```

Listing 11: checkTopic() gets called when the publisher wishes to create a new topic

The code in listing 12 shows the publishMessage() method. This is called when the publisher wants to publish a message for a topic. The user is prompted to enter the topic name they wish to publish for. This string is then used to check if that topic exists in the topics list, if it doesn't exist an error message is displayed. If the topic does exist the user is prompted to enter the message they wish to publish and the entered string is then bundled into the correct number of packets and sent to the broker.

```
private void publishMessage() throws InterruptedException {
    String topic = terminal.readString("Enter the topic to Publish for: ");
    Integer topicNumber = null;
    for(Integer key : topicNumbers.keySet()) {
            if(topicNumbers.get(key).equals(topic))
                    topicNumber = key;
    }
    if (topicNumber == null) {
            terminal.println("This topic does not exist.");
    } else {
            String message = terminal.readString
         ("Please enter the message that you would like to publish: ");
            DatagramPacket[] packets =
         createPackets(Constants.PUBLICATION, topicNumber, message, dstAddress);
            sendPackets(packets);
            terminal.println("Packet sent");
    }
}
```

Listing 12: publishMessage() method, Handles user input and message publication

The code in listing 13 shows the start() method. This is the main loop of the publisher, it asks the user what action they wish to carry out. It will carry out the current task and when complete will the give the user options to do more operations.

```
public synchronized void start() throws Exception {
    while (true) {
            String startingString = terminal.readString(
                 "Please choose an operation to carry out:\n" +
         "1. Create a topic\n" + "2. Publish a message\n");
            if (startingString.toUpperCase().contains("1")) {
                    checkTopic();
            } else if (startingString.toUpperCase().contains("2")) {
                    publishMessage();
            }
    }
}
```

Listing 13: start() method of the publisher. Loops waiting for input

The code in listing 14 shows the onReceipt() method, this handles what to do when each type of packet is received.

```
public synchronized void onReceipt(DatagramPacket packet) {
    this.notify();
    byte[] data = packet.getData();
    if (data[0] == Constants.ACK) {
            // print the message in the acknowledgement
            terminal.println(processMessageFromPacket(packet));
    } else if (data[0] == Constants.CREATION) {
            int topicNumber = data[3];
            addTopicToList(topicNumber, processMessageFromPacket(packet));
    }
}
```

Listing 14: onReceipt() method, handles packets received on the publishers port

3.3 Subscriber

Most of the initialization for the Subscriber is similar to the Publisher. Its constructor initializes the socket with the correct port number, it also creates the sorted treemap that is used as a cache for incoming packets.

The code in listing 15 shows the start() method for the subscriber. This continually loops waiting for input from the user. When the user picks and option, that task is carried out and then the user has the option to pick another task to carry out when it is complete. If the user wished to subscribe, the subscribe() method gets called. Similarly if the user wished to unsubscribe from a topic, the unsubscribe() method is called.

```
public synchronized void start() throws Exception {
    while (true) {
            if (state == Constants.SUB_DEFAULT_STATE) {
                    String command = terminal.readString
                ("Please␣choose␣an␣action␣to␣be␣carried␣out:\n␣"
                + "1.␣Subscribe␣to␣a␣topic\n" + "2.␣Unsubscribe␣from␣a␣topic\n"
                            + "3.␣Wait␣for␣topics\n");
                    if (command.contains("1"))
                            subscribe();
                    else if (command.contains("2"))
                            unsubscribe();
                    else if (command.contains("3")) {
                            terminal.println("waiting␣for␣publications");
                    }
                    this.wait();
            } else if (state == Constants.SUB_AWAIT_NEW_PACKET) {
                    this.wait();
            }
    }
}
```

Listing 15: start() method, handles user interaction

The code in listing 16 shows the subscribe() method. This prompts the user to enter the name of a topic they wish to subscribe to. This string is then bundled into a packet and sent to the broker, where the broker will check if the topic exists, if not it will send back an ack with contents telling the subscriber that the subscription failed. If the topic does exist the broker will add the subscribers port number to the list of port numbers for that topic and send an ack with contents success.

```
public synchronized void subscribe() {
    byte[] topic = (terminal.readString
    ("Please␣enter␣a␣topic␣to␣subscribe␣to:␣")).getBytes();
    byte[] data = createPacketData
    (Constants.SUBSCRIPTION, 0, 0, 0, topic);
    terminal.println("Sending␣packet...");
    DatagramPacket packet = new DatagramPacket(data, data.length, dstAddress);
    try {
            socket.send(packet);
    } catch (IOException e) {
            e.printStackTrace();
    }
    terminal.println("Packet␣sent");
}
```

Listing 16: subscribe() method, takes input from the user and sends a subscribe packet to the broker

The code in listing 17 shows the unsubscribe() method. This prompts the user to enter the name of a topic they wish to unsubscribe from. This string is then bundled into a packet and sent to the broker, where the broker will check if the topic exists, if not it will send back an ack with contents telling the subscriber that the unsubscription failed. If the topic does exist the broker will remove the subscribers port number from the list of port numbers for that topic and send an ack with contents success.

```
public synchronized void unsubscribe() {
    byte[] topic = (terminal.readString
    ("Please enter a topic to unsubscribe from: ")).getBytes();
    byte[] data = createPacketData
    (Constants.UNSUBSCRIPTION, 0, 0, 0, topic);
    terminal.println("Sending packet...");
    DatagramPacket packet = new DatagramPacket(data, data.length, dstAddress);
    try {
            socket.send(packet);
    } catch (IOException e) {
            e.printStackTrace();
    }
    terminal.println("Packet sent");
}
```

Listing 17: unsubscribe() method, takes input from the user and sends a unsubscribe packet to the broker

The code in listing 18 shows the flushBuffer() method. When the subscriber receives a packet it saves it to its buffer. In this method we check if the packet received is the end of a publication, if so it will change the state of the subscriber and flush the buffer to the terminal.

```
private void flushBuffer(DatagramPacket packet) {
                if (lastPacket(packet)) {
                        state = Constants.SUB_DEFAULT_STATE;
                        printPublication();
                } else {
                        state = Constants.SUB_AWAIT_NEW_PACKET;
                }
        }
```

Listing 18: flushBuffer() method

The code in listing 19 shows the printPublication() method. This takes the ordered packets from the received publication treemap and adds them to a string, which is then printed to the terminal.

```
public void printPublication() {
                for (Integer key : receivedPublication.keySet()) {
                        DatagramPacket packet = receivedPublication.get(key);
                        String message = processMessageFromPacket(packet);
                        buffer += message;
                }
                terminal.println(buffer);
                buffer = "";
        }
```

Listing 19: printPublication() method, prints out a publication to the terminal

The code in listing 20 shows the onReceipt() method, this handles what to do when each type of packet is received.

```
public synchronized void onReceipt(DatagramPacket packet) {
                this.notify();
                byte[] data = packet.getData();
                if (data[0] == Constants.ACK) {
                  terminal.println("ACK: " + processMessageFromPacket(packet));
                } else if (data[0] == Constants.PUBLICATION) {
                  receivedPublication.put((int) data[1], packet);
                  flushBuffer(packet);
                }
        }
```

Listing 20: onReceipt() method, handles packets received on the subscribers port

3.4 Broker

The code in listing 21 shows the constructor for the broker, this sets up its socket address and port number which is based on a constant. It also allocates memory for two hashmaps, the subscriberMap which will store all of the subscriber port numbers for each topic and the topicNumbers map which will store all of the existing topic names along with their corresponding topic number.
The code in listing 25 shows the createTopic() method. This gets called when the publisher sends a topic creation request and the topic does not exist already. The topic name is passed in as a parameter. An arraylist of subscriber socket numbers is created to give to the topic. The topic is then added to the subscriberMap and topicNumbers hashmaps and is given its own topic number.

```
private void createTopic(String topicName) {
                ArrayList<Integer> socketNumbers = new ArrayList<Integer>();
                subscriberMap.put(topicName, socketNumbers);
                topicNumbers.put(topicNumbers.size(), topicName);
                terminal.println("Topic " + topicName + " was created.");
        }
```

Listing 21: onReceipt() method, handles packets received on the publishers port

The code in listing 22 shows the checkTopic() method, this checks if a topic already exists. If it does it sends a reply to the publisher containing the topic number. If the topic does not exist the createTopic() method is called to add it to the list of topics. Then a reply is sent to the publisher containing the topic number of the newly created topic. This allows the publisher to update its list of topics and corresponding topic numbers.

```
private void checkTopic(String topicName, DatagramPacket packet) {
                Integer topicIndex = null;
                for(int index = 0; index < topicNumbers.size(); index++) {
                        if(topicNumbers.get(index).equals(topicName));
                                topicIndex = index;
                }
                if(topicIndex == null) {
                        createTopic(topicName);
                        sendTopicNumber(topicNumbers.size()-1, packet);
                }
```

```
                else {
                        sendTopicNumber(topicIndex, packet);
                }
        }
```

Listing 22: onReceipt() method, handles packets received on the publishers port

The code in listing 23 shows the sendTopicNumber() method. This method gets invoked after the publisher makes a creation request. It takes a topic number as a parameter along with a packet which is used to get the correct port number to send the response to. It creates a new creation packet containing the topic number of the requested topic with the destination being the publisher. It then send this packet to the publisher.

```
private void sendTopicNumber(int topicNumber, DatagramPacket receivedPacket) {
   byte[] newPacketData = createPacketData(Constants.CREATION,
       0, 0, topicNumber, processMessageFromPacket(receivedPacket).getBytes());
   DatagramPacket newPacket = new DatagramPacket
         (newPacketData, newPacketData.length);
   newPacket.setSocketAddress(receivedPacket.getSocketAddress());
   try {
            socket.send(newPacket);
   } catch (IOException e) {
            System.out.println("Broker failed to send" +
                "confirmation message to publisher.");
   }
}
```

Listing 23: onReceipt() method, handles packets received on the publishers port

The code in listing 24 shows the sendAck() method. This method sends an acknowledgment to any received communication. It takes a message to be placed in the ack packet if desired and also the received packet socket information can be obtained about the sender. The necessary information is extracted from the received packet and new ack packet is created containing the optional message. This ack packet is then sent to the destination.

```
private void sendAck(String message, DatagramPacket receivedPacket) {
                byte[] data = receivedPacket.getData();
                byte[] messageInBytes = message.getBytes();
                int packetNumber = data[1];
                byte[] confirmationPacketContent = createPacketData
                 (Constants.ACK, packetNumber, 0, 0, messageInBytes);
                DatagramPacket confirmation = new DatagramPacket
                 (confirmationPacketContent, confirmationPacketContent.length);
                confirmation.setSocketAddress
                 (receivedPacket.getSocketAddress());
                try {
                        socket.send(confirmation);
                } catch (IOException e) {
                        System.out.println("Broker"+
                    "failed to send confirmation message to publisher.");
                }
        }
```

Listing 24: onReceipt() method, handles packets received on the publishers port

The code in listing 25 shows the publishMessage() method. This method takes a publication packet from the publisher and forwards it to any subscribers for that topic. The topic number is extracted from the packet

and is used as a key to find that topic in the topicNumbers hashmap. If the topic exists, the topic name is used as a key to find the subscribers to that topic. After this a for-loop is used to increment through the subscribers and forwards the publication to each one.

```
private void publishMessage(DatagramPacket packet) {
    byte[] message = packet.getData();
    int topicNumber = message[3];
    if (topicNumbers.containsKey(topicNumber)) {
        String topicName = topicNumbers.get(topicNumber);
        if (subscriberMap.containsKey(topicName)) {
            ArrayList<Integer> topicSubscribers = subscriberMap.get(topicName);
            for (int index = 0; index < topicSubscribers.size(); index++) {
                InetSocketAddress subAddress = new InetSocketAddress
                (Constants.DEFAULT_DST_NODE, topicSubscribers.get(index));
                DatagramPacket messagePacket = new DatagramPacket(message,
                message.length, subAddress);
                try {
                    socket.send(messagePacket);
                } catch (IOException e) {
                    System.out.println ("Broker failed to send"
                    + "confirmation message to publisher.");
                }
            }
        }
    }
}
```

Listing 25: onReceipt() method, handles packets received on the publishers port

The code in listing 26 shows the subscribe() method. When the broker receives a subscribe packet, the onReceipt() method will call the subscribe() method, passing the received packet as a parameter. The topic name is then extracted from the message portion of the packet and used as a key to check if such a topic exists. If the topic exists the subscribers port number gets added to the hashmap of subscribers for that topic. Depending on the outcome, an acknowledgment will be sent to the subscriber letting them know the outcome.

```
private void subscribe(DatagramPacket packet) {
    String topic = processMessageFromPacket(packet);
    if (subscriberMap.containsKey(topic)) {
            ArrayList<Integer> topicSubscribers = subscriberMap.get(topic);
            if(!topicSubscribers.contains(packet.getPort())){
                topicSubscribers.add(packet.getPort());
            }
            sendAck("Success", packet);
    } else {
            sendAck("Topic Doesnt Exist", packet);
    }
}
```

Listing 26: onReceipt() method, handles packets received on the publishers port

The code in listing 27 shows the unsubscribe() method. This follows the same premise as the subscribe() method, except rather than adding a port number to the list of subscribers, this method removes the port number from the list of subscribers if it exists.

```
private void unsubscribe(DatagramPacket packet) {
    String topic = processMessageFromPacket(packet);
    if (subscriberMap.containsKey(topic)) {
            ArrayList<Integer> topicSubscriptions = subscriberMap.get(topic);
            for (int index = 0; index < topicSubscriptions.size(); index++) {
                    if (topicSubscriptions.get(index) == packet.getPort()) {
                            topicSubscriptions.remove(index);
                    }
            }
            sendAck("Success", packet);
    } else {
            sendAck("No Subscription Exists", packet);
    }
}
```

Listing 27: onReceipt() method, handles packets received on the publishers port

The code in listing 28 shows the onReceipt() method for the broker. This is the longest onReceipt() method as the broker has the most functionality. When each packet is received its packet type gets extracted from the header. Depending on the type of packet the correct branch of the if statement gets executed.

```
public synchronized void onReceipt(DatagramPacket packet) {
    try {
            this.notify();
            byte[] data = packet.getData();
            System.out.println(printPacketContent(packet));
            if (data[0] == Constants.CREATION) {
                    terminal.println("Creation Received From The Publisher");
                    String topic = processMessageFromPacket(packet);
                    checkTopic(topic, packet);
                    sendAck("The topic " + topic + " was created.", packet);
            } else if (data[0] == Constants.PUBLICATION) {
                    terminal.println("Publication Received From The Publisher");
                    publishMessage(packet);
                    sendAck("Message has been published", packet);
            } else if (data[0] == Constants.SUBSCRIPTION) {
                    terminal.println("Subscription recieved from Subscriber.");
                    subscribe(packet);
            } else if (data[0] == Constants.UNSUBSCRIPTION) {
                    terminal.println("Unsubscription recieved from Subscriber.");
                    unsubscribe(packet);
            }
    } catch (Exception e) {
            e.printStackTrace();
    }
}
```

Listing 28: onReceipt() method, handles packets received on the publishers port

3.5 User Interaction

The user interacts with the publish subscribe system through the terminal. this was provided as part of the tcdIO library. A terminal has some of the functionality that is included in the console, such as reading text that has been written in the terminal and printing text out to the terminal. This allows for each party in the system to have its own unique terminal window rather than sharing the same console. This is very convenient when testing on the same machine as it allows for separation of messages and also makes the flow of the program much more apparent and easy to follow. It also greatly aids debugging.

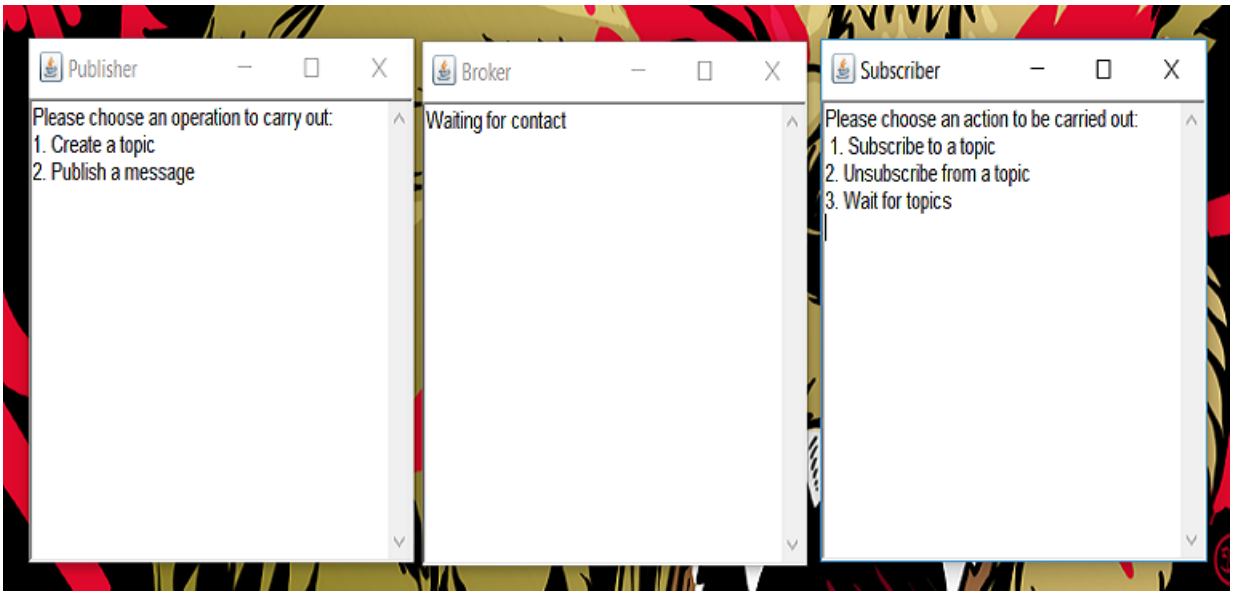

Figure 2: Shows three terminal windows, one for each node in the system

Each node has a different interface based on the functionality it provides. For example, the publisher will ask the user if they would like to create a topic or if they would like to publish a message. The broker has no user input but does display messages based on some of the operations that it is carrying out. The subscriber gives the user the option to subscribe to a topic, unsubscribe from a topic or to wait for messages.

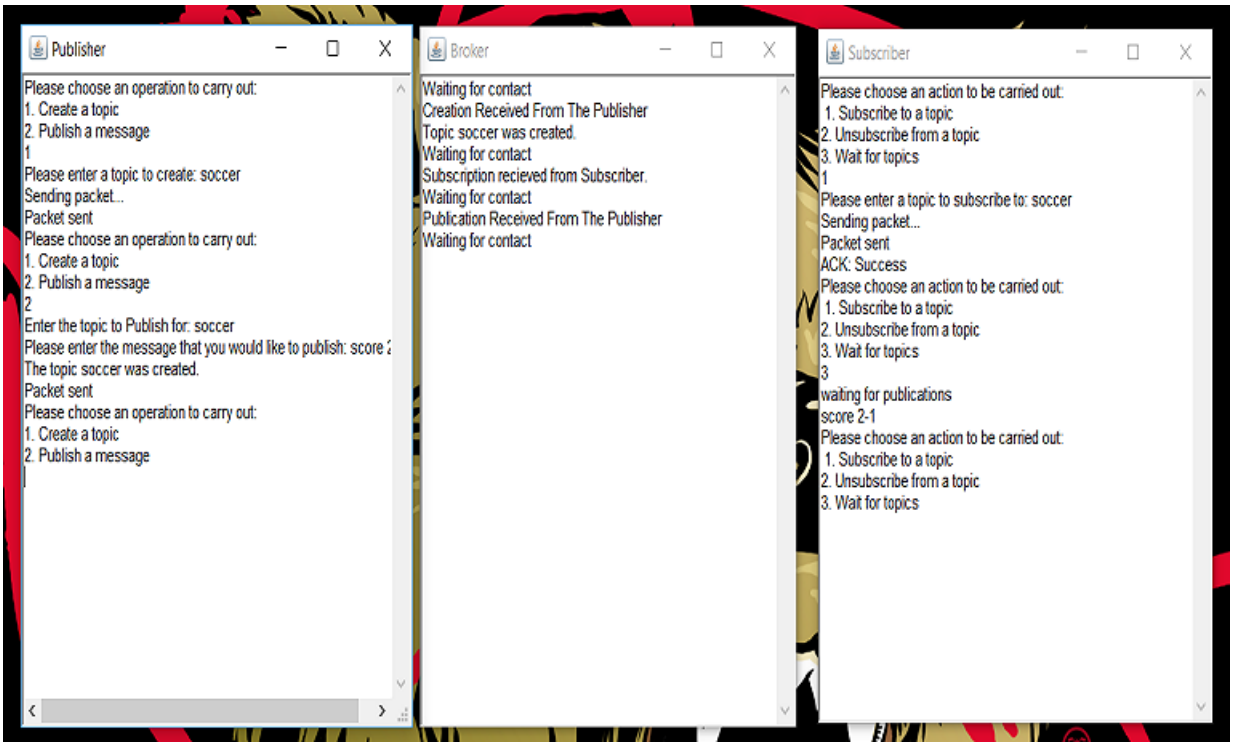

Figure 3: Shows three terminal windows and the normal operation flow of the system

3.6 Packet Design

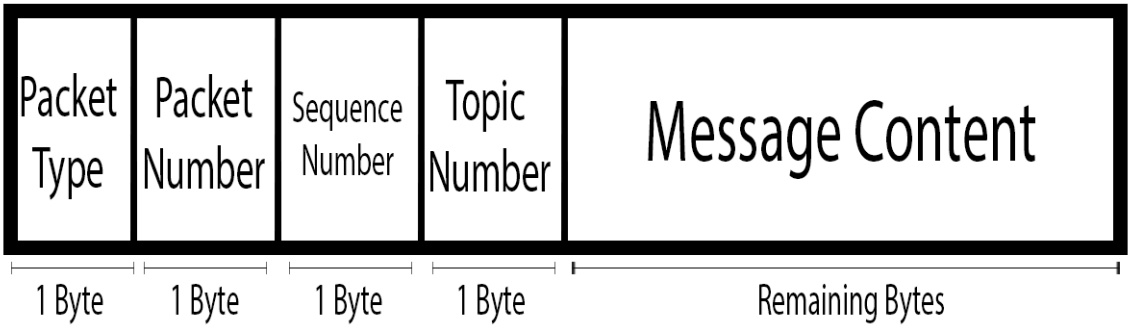

Figure 4: Shows the packet layout

The packets that I have designed are very simplistic but also have the facility to allow for expansion in the future if i had time to implement an Automatic Repeat Request(ARQ) system. Each packet is just an array of bytes. The byte 0 is the packet type, this will be based on a constant so all stations know what each packet type means. Examples of a few packet types that were implemented include publication, subscription and acknowledgment.

The byte 1 contains the packet number, this is useful for flow control and could also be used in an ARQ implementation. A single byte = 8 bits which allows for 2 to the power of 8 packet numbers which is more than enough for this example.

Byte 2 is the sequence number, this is the total amount of packets in a transmission, again this would

be very useful for an ARQ technique as it would make framing and re-sending packets very simple. In absence of this, it still provides useful information as to how many packets are expected, and also for sorting packets when they are received. e.g. in the subscriber, if the packet number of the packet received is the same as the sequence number, we know that this is the last packet in the current transmission and we can print the received message to the terminal and clear the message buffer.

Byte 3 includes the topic number that this packet involves. This is extremely useful for this application as it allows for quick and simple identification of packets, for example if the publisher sends a packet of type publication, with a topic number of 3, the broker can quickly parse the topic number from the packet and send the publication to all the subscribers for topic number 3.

The remaining bytes of the packet are used for the message that is being sent. The message which is a string, is converted to an array of bytes based on the ascii character set. This is then bundled into the message portion of the packet and null terminated. If the message is greater than the available size in a single packet, it gets broken up into multiple packets that are then sent.

3.7 Acknowledgments

For this assignment I have assumed that the connection is perfect and that no packets will be dropped or corrupted during the transmission process. As a result of this I have not implemented any ARQ technique such as stop-and-wait, go-back-n or selective repeat. This simplified my code substantially and made it much easier to display the functionality and premise of a publish-subscribe system.

To implement an ARQ system would require some method of tracking timers for packets as we need to have a facility for packets to time out. It would also require some sort of framing and window system to send packets and receive them. All of these extra features would have made the code more complex but would have made the system more robust.

If the timers had been implemented I would have done so using time-stamps. Each packet could be bundled into a frame to be sent, each frame would contain a packet and a timer. Based on the value in the timer and using the current time it could be determined if the packet has timed out, and if so resent.

3.8 Extra Features

The two main extra features that were implemented include dynamic splitting of messages into separate packets and synchronized topic numbers between the publisher and broker.

If a message is too big to fit into one packet, rather than limiting the size of the message, the message is divided into sections and each section is put into a packet and sent. This feature is implemented in the node class using the createPackets() method. All the packet information is passed into this method, then the max message size is determined and if this is less than the length of the message then using a for-loop the message gets converted to an array of bytes and then is divided accordingly. This makes for easy packet and sequence number allocation and returns an array of packets that can be sent to the destination.

If more than 1 publisher was being used then I encountered a problem in my testing where topic names and topic numbers would not synchronized between the publisher and the broker. Take an example with 2 publishers, publisher 1 could have topic number 1 = soccer, and inform the broker. But if publisher 2 creates topic number 1 = rugby, it will try to inform the broker which will just ignore the message, as topic number 1 already exists. Therefore there would be difficulty in publishing messages for these topics. To avoid this problem, i remodeled the system so the broker holds the "master list" of topic numbers. When a publisher wishes to create a topic, it sends a creation packet to the broker. The broker then ensures that this topic does not exist, if it does, it will return the topic number, if it does not exist, it will allocate it a new topic number and return this new topic number. This way all topics and topic numbers are synchronized between the broker and multiple publishers.

I also implemented piggybacking on the acknowledgments. Normally the acknowledgments would just contain the sequence number of the packet that it successfully received. Piggybacking allows for more information to be sent, I used the message portion of the Acknowledgment packet to store a short message that would indicate what had occurred. For example when a subscriber sends a subscribe packet to the broker, the broker will send an acknowledgment to the subscriber that the packet was received and in the message portion of the packet a small sentence such as "Success". This allows the subscriber to have some indication of what happened on the other end of the communication.

4 Discussion

Some of the strengths of this program are the straight forward design, which is easy to read and understand. I think it captures the important elements of a publish/subscribe system. It can also handle more than one publisher and subscriber which allows for larger systems.

The flow control could be improved to include an implementation of ARQ but we were told that it was not required for this assignment so I didn't focus on it; it would have also complicated the code and this would have made it more difficult to convey the workings of the publish/subscribe system. As a result of this choice packets can get lost, in that case the system will just hang as one of the nodes will be waiting for an acknowledgment and won't function as intended.

Another limitation of this implementation is due to the use of the tcdIO terminal. The terminal will "hang" or wait for user input at specific points in time. Take the subscriber as an example. When the subscriber it booted up and the initial menu is displayed, the terminal is waiting for the user to input an option, the code will not continue until this happens. The subscriber does have a listener thread which can successfully receive packets while the terminal is waiting for input, but unfortunately can't display these packets to the terminal until the user has selected an option. That is why there was a necessity to add option 3 to wait for topics, this leaves the terminal in a state where it is ready to be written to and thus can print received messages. This limitation in design means that while waiting for packets the user cannot subscribe or unsubscribe and must wait for something to be received before it can interact again.

There is a problem with the implementation of forwarding messages from the publisher to the subscriber. When the broker receives a publication packet from the publisher, it forwards it to the subscriber straight away, this has the potential to go wrong as if the broker misses a packet from the publisher due to interference then there is no opportunity for re-transmission to the subscribers as the packets have not been cached in any way. A better implementation would be if the broker cached the publication and ensured that it had all been correctly received from the publisher. Once this was completed then forward the publication to each subscriber. This would be much more robust than the current approach.

At the beginning of this assignment I spent quite some time figuring out how to use wireshark and docker, this was well worth the time that I spent on it even though as the assignment progressed I moved further away from both. I found running the code on the same machine using localhost to be much more convenient than setting up docker each time, this meant that I could concentrate on the code and get working faster. I ended up using my own packet printing function for debugging rather than capturing packets using wireshark. Although the experience using both software tools have been invaluable.

To add more Publishers or Subscribers I manually changed the port numbers before starting them, this could have been implemented using the terminal to choose the port numbers.

5 Summary

Here is a small summary of what was covered in this report:

A bit of background on a publish/subscribe system, explains what it is and how it works.

Outlined the Steps involved in solving the problem and how each element of the design was approached.

Showed the operation of the system through explanations supported by code snippets and screenshots.

Explained any of the assumptions and short comings of this implementation.

Discusses the extra features implemented and all the points that could have been improved upon.

6 Reflection

Overall I was quite happy with how the assignment turned out as a whole, but I feel like time management is something I really need to improve upon. This assignment took well over 30 hours in total, this was partially due to the fact that I launched into it too quickly before having the relevant knowledge. This resulted in lots of wasted time redesigning code and fixing mistakes. Using the weekly labs worked well for setting deadlines for features and also allowed me to get help when I needed it. Going forward in the next assignment I hope to improve my time management and also do more background research before I launch into writing code. When writing the code for this assignment I done so in stages, slowly implementing new features one at a time. This approach worked well for debugging and I will use this going forward. I will change the way I approach the report for the next assignment. Including code snippets for all of my code took a long time, for the next assignment I will include code snippets for interesting pieces of code and also code which may be hard to explain. For this report I made a draft in Microsoft word first, rather than making a draft then moving it over to Latex, I will write the report from the beginning in Latex now that I am comfortable using it.

Included in the source files is an implementation I abandoned as is became too complicated and had too many bugs. This implementation includes queues, sliding windows and the early stages of a go-back-n system. It does not function as multiple bugs got introduced but it does demonstrate how I would have gone about Implementing an ARQ technique.

7 References

-CS2031 notes and example documents.
-Java Documentation - www.oracle.com
-Message Queuing Telemetry Transport information from - mqtt.org