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

CSU33031 Computer Networks

IoT Publish/Subscribe Protocol

Prathamesh Sai
3rd year Integrated Computer Science
Student ID: 19314123

School of Computer Science and Statistics,
Trinity College Dublin

Contents

1 Introduction

This assignment is focused on learning about protocol development and the information that is kept in a header to support functionality of a protocol. The aim for this protocol is to provide a publish/subscribe mechanism for processes based on UDP datagrams. We have 2 minimum requirements for this assignment.

1. The issuing of sensor data to a number of subscribers (dashboard subscribes to a broker, and sensors publish their data).
2. The issuing of instructions to actuators (actuators subscribe to a broker, and a dashboard sends instructions to the actuators).

The communication between the processes is realized using UDP sockets and Datagrams. My implementation is written in Java and it can be run on Docker, meaning that my protocol can connect components located at a number of hosts.

2 Theory of Topic

I will now talk about the theory behind the publish/subscribe protocol. This will allow you to understand some background information that will be useful in understanding my implementation of the protocol.

2.1 Sequence Number

The sequence number allows us to know the order in which packets have been sent, hence allowing us to construct the data in the right order regardless of any packet loss/reordering.

2.2 Header

The header is a piece of data that is attached to the start of a packet. The header contains several pieces of metadata such as the type packet it is attached to, the sequence number, the topic number, etc.

2.3 Subscriber

A subscriber can subscribe to topic from a broker, and will then receive all messages published to that topic.

2.4 Publisher

A publisher can create and publish to a topic, sending this information to a broker. It does not directly communicate with a subscriber, but only does so indirectly with the help of a broker.

2.5 Broker

A broker manages communication between subscribers subscribed to it, and publishers publishing to a topic. In simple terms, A broker receives information from publishers and forwards this information to all the relevant subscribers.

2.6 MTU

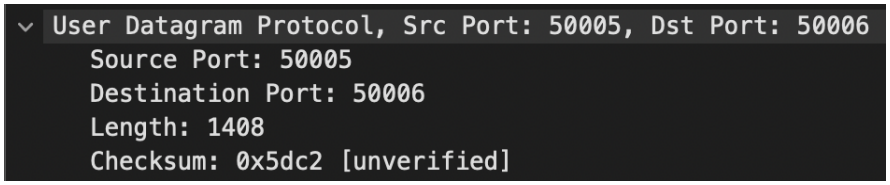
The MTU (Maximum Transmission Unit) is determined by the maximum frame size. The MTU for the Ethernet protocol is 1500 bytes. For this assignment, it was important for me to determine a maximum packet size for the protocol so that it could accommodate relatively small Maximum Transmission Units.

2.7 IP

IP (Internet Protocol) is a networks layer communication protocol. There are two versions used today, IPv4 and IPv6. The network layer is in charge of moving packets from a source to a destination, hence it is important to have an IP address in the IP header. The IP header contains the IP address of the destination address and the source address. All IP addresses are 32 bits long in IPv4, while being 128 bits long in IPv6. In the java.net library, an IP address is represented by an InetAddress object, using either IPv4 or IPv6.

2.8 UDP

UDP (User Datagram Protocol) is a transport layer communication protocol. It is commonly used for applications that are "lossy" (i.e. can handle some packet loss), such as streaming audio and video. In a UDP header, there is a 16-bit source port number, 16-bit destination port number, and a 16-bit checksum value.



UDP is able to support multiple network applications at the same time, allowing applications to send and receive network data simultaneously. It does this by using application level addressing, otherwise known as port numbers. Data from different applications are multiplexed at a sending device and demultiplexed at the receiving device, both using port numbers. The checksum value on the other hand is used for error control but it is optional in UDP. UDP is a connectionless protocol meaning no connection needs to be established between the source and destination before you transmit data, and the header is only 8 bytes long with no flow control and optional error control as mentioned above; UDP can be seen as an unreliable protocol in a sense.

3 Implementation

I will now talk about how my publish/subscribe implementation works with the theory I stated above. I have written my implementation in Java. I used OpenJDK-8 as the Terminal functionality in tcdlib.jar on blackboard was not compatible with OpenJDK-16.

3.1 Node.java

The Node class is an abstract class that is extended by the Publisher, Subscriber, and the Broker. We were provided a generic Node.java file on Blackboard for the Docker Walkthrough, which I extended further to suit my implementation. This class contains a nested Listener class which constantly listens for any incoming packets, calling the onReceipt function on arrival.

```
//These are the packet size which stays constant, and the default dst
string used for InetSocketAddress.
static final int SIZE_OF_PACKET = 1400;
static final String DEFAULT_DST = "Broker";
/*
 * This function is Abstract so it can be adjusted to different classes
 such as Subscribers,
 * Publishers, and Brokers. Hence, different classes can behave
 differently when they receive packets.
*/
public abstract void onReceipt(DatagramPacket packet);

/**
 * This class listens for incoming packets on a Datagram socket and
 tells appropriate
 * receivers about such incoming packets.
 */
  class Listener extends Thread {
  //Stating to the listener that the socket has been initialized.
  public void go() {
    latch.countDown();
  }

  //Listen for incoming packets and inform receivers.
  public void run() {
    try {
      latch.await();
      // Infinite loop that attempts to receive packets, and notify
      receivers.
      while (true) {
        DatagramPacket packet = new DatagramPacket(new byte[PACKETSIZE],
        PACKETSIZE);
        socket.receive(packet);
        onReceipt(packet);
      }
    } catch (Exception e) {
      if (!(e instanceof SocketException))
        e.printStackTrace();
    }
```

```
    }
  }
}
```

This onReceipt function is abstract, so any class that extends Node can deal with the incoming packets in a unique way, which is what helped me develop my solution. The Node class contained constants such as SIZE_OF_PACKET which was set to 1400 bytes so my implementation would work well with small MTU's; I implemented a fixed length packet. DEFAULT_DST which is the IP address for the nodes, called "Broker" as the communication of the Publisher and Subscriber is closely managed by the Broker. The nodes are given are given unique port numbers, since it is vital that we have different port numbers for different nodes to work well using UDP.

```
static final int SUBSCRIBER_1_PORT = 50002;
static final int SUBSCRIBER_2_PORT = 50003;
static final int PUBLISHER_1_PORT = 50004;
static final int PUBLISHER_2_PORT = 50005;
static final int BROKER_PORT = 50006;
```

The UDP header contains the source and destination port and the IP header contained the source and destination IP addresses as mentioned in the Theory of Topic (2.8 & 2.9). Therefore, I had to implement the array of bytes in the DatagramPacket, which are the packets used in this protocol. There are different types of packets in my protocol.

```
//These are the types of packets for this project.
static final byte ACK = 0;
static final byte CREATE_TOPIC = 1;
static final byte PUBLISH = 2;
static final byte SUBSCRIBE = 3;
static final byte UNSUBSCRIBE = 4;
static final byte MESSAGE = 5;
```

The packets in my protocol are created with each byte representing a different piece of information that is critical for my implementation to work. Byte 0 represents the type of packet in question, byte 1 represents the sequence number, and bytes 2 to 5 represent the topic number. I wanted to implement error control as a bonus to this assignment so I added sequence numbers at the start but due to a lack of time I was unable to properly implement them so I just set it to 0. The remaining bytes are the payload, which is simply a string of characters carrying a message.

```
/**
 * This function creates packets by taking in the type of packet, topic
 number,
 * message and destination address - The message must fit in one packet
 for it to work.
 */
protected DatagramPacket[] createPackets(int typeOfPacket, int
topicNumber, String message, InetSocketAddress dstAddress) {
  int numberOfPacketsForMessage = 0;
```

```
    int sizeOfMessage = SIZE_OF_PACKET - 6;
    int offset = 0;

    //Convert the String message to an array of bytes.
    byte[] temporaryArray = message.getBytes();
    byte[] messageArray = new byte[temporaryArray.length];
    for (int i = 0; i < temporaryArray.length; i++) {
      messageArray[i] = temporaryArray[i];
    }

    //Calculate the number of packets in the message.
    for (int messageLength = messageArray.length; messageLength > 0;
    messageLength -= sizeOfMessage) {
      numberOfPacketsForMessage++;
    }

    //Create DatagramPacket to return with the size of the number of
    packets required from the message.
    DatagramPacket[] packets = new
    DatagramPacket[numberOfPacketsForMessage];

    //Create packet data and add to DatagramaPacket packets variable.
    for (int sequenceNumber = 0; sequenceNumber <
    numberOfPacketsForMessage; sequenceNumber++) {
      byte[] offsetMessage = new byte[sizeOfMessage];
      for (int j = offset; j < offset + messageArray.length; j++) {
        offsetMessage[j] = messageArray[j + offset];
      }
      byte[] data = createPacketData(typeOfPacket, sequenceNumber,
      topicNumber, offsetMessage);
      DatagramPacket packet = new DatagramPacket(data, data.length,
      dstAddress);
      packets[sequenceNumber] = packet;
      offset += sizeOfMessage;
    }
    return packets;
  }

  /**
   * This function creates packet data by creating an array of bytes for a
   * Datagram Packet.
   * HEADER INFORMATION:
   * byte[0] = typeOfPacket
   * byte[1] = sequence number for GO-BACK-N protocol.
   * byte[2->5] = topic number
   * PAYLOAD:
   * byte[6->endOfArray] = remaining bytes for the message.
   */
  private byte[] createPacketData(int typeOfPacket, int sequenceNumber,
  int topicNumber, byte[] message) {
    byte[] packetData = new byte[SIZE_OF_PACKET];
    packetData[0] = (byte) typeOfPacket;
    packetData[1] = (byte) sequenceNumber;

    //Range of bytes 2->5 in packetData is allocated for the topic number.
```

```
    ByteBuffer byteBuffer = ByteBuffer.allocate(4);
    byte[] topicNumberArray = byteBuffer.array();
    for (int i = 0; i < 4; i++) {
      packetData[i + 2] = topicNumberArray[i];
    }

    //Any remaining bytes in packetData is used for the message.
    for (int i = 0; i < message.length && i < SIZE_OF_PACKET; i++) {
      packetData[i + 6] = message[i];
    }
    return packetData;
  }
```

Although I was not able to implement error control I made sure to implement acknowledgements from the beginning of my attempt at this assignment as I knew it would be helpful for different Nodes to receive an acknowledgement every time they sent a packet. There is also an acknowledgement given to Nodes if their subscription, publication, or topic creation was successful.

```
  /*
   * This function sends an acknowledgement (DatagramPacket) through the
   socket,
   * and the packet is set with the appropriate acknowledgement type
   before sending it.
   */
  protected void sendAck(DatagramPacket packetSentHere, Terminal terminal)
  {
    //Convert received packet to byte array to set type to acknowledgement.
    byte[] data = packetSentHere.getData();
    ProtocolUtility.setType(data, ACK);
    try {
      //Convert back to DatagramPacket after type has been set and sent
      through the socket.
      socket.send(new DatagramPacket(data, data.length,
      packetSentHere.getSocketAddress()));
      terminal.println("ACK has been sent!");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

3.2 ProtocolUtility.java

The ProtocolUtility class contains utility functions that are used throughout my code. I made a dedicated class for it, so that my code is cleaner, more understandable, and if I wanted to extend my implementation in the future, I don't need to repeat code. The utility class also contains a HashMap called topicNumbersToNamesMap which maps topic numbers (key) to topic names (value). This is agreed with the broker, and is contained in the utility class so multiple publishers can use the same global variable, rather than multiple publishers having their own version in their class.

```
import java.nio.ByteBuffer;
import java.util.HashMap;
import java.util.Map;

public final class ProtocolUtility {

  private ProtocolUtility() throws Exception {
    throw new Exception();
  }
  //Global variable for publishers that maps topic numbers to topic names,
  map agreed with Broker.
  static Map<Integer, String> topicNumbers = new HashMap<Integer,
  String>();

  protected static int getType(byte[] data) {
    return data[0];
  }

  protected static void setType(byte[] data, byte type) {
    data[0] = type;
  }

  protected static int getSequenceNumber(byte[] data) {
    return data[1];
  }

  protected static int getTopicNumber(byte[] data) {
    byte[] intArray = new byte[4];
    for (int i = 0; i < intArray.length; i++) {
      intArray[i] = data[i + 2];
    }
    return ByteBuffer.wrap(intArray).getInt();
  }

  protected static String getMessage(byte[] data) {
    byte[] messageArray = new byte[data.length - 6];
    for (int i = 0; i < messageArray.length && data[i + 6] != 0; i++) {
      messageArray[i] = data[i + 6];
    }
    String message = new String(messageArray).trim();
    return message;
  }
}
```

3.3 Broker.java

The Broker class is used for dealing with communication between the Subscriber and the Publisher. I have two HashMaps which are used to implement this functionality, and I use the terminal class to output information. The first map is called topicNumbersToTopicNamesMap, and it maps topic numbers (key) to topic names (value) to have a unique topic number for every name for a topic. The second map is called topicsToSubscribersMap, and it maps topic names (key) to a useful list of port numbers of the

subscribers subscribed to it (value) to ensure my implementation works with multiple subscribers.

```
/*
 * This function creates a topic by using data passed in from a creation
 packet.
 */
private boolean createTopic(byte[] data) {
  ArrayList<InetSocketAddress> socketNumbers = new
  ArrayList<InetSocketAddress>();
  String topicName = ProtocolUtility.getMessage(data);
  if (!topicsToSubscribersMap.containsKey(topicName)) {
    topicsToSubscribersMap.put(topicName, socketNumbers);
    int topicNumber = ProtocolUtility.getTopicNumber(data);
    topicNumbersToTopicNamesMap.put(topicNumber, topicName);
    terminal.println("A topic called " + topicName + " was created!");
    return true;
  }
  return false;
}

/*
 * This function sends a message as a Datagram Packet given the message
 as a String and the destination
 * address to where the message will be sent.
 */
private void sendMessage(String message, SocketAddress socketAddress) {
  DatagramPacket packet = createPackets(MESSAGE, 0, message,
  (InetSocketAddress) socketAddress)[0];
  try {
    socket.send(packet);
    terminal.println("The Broker sent a message: " + message);
  } catch (IOException e) {
    e.printStackTrace();
    terminal.println("The Broker failed to send a message: " + message);
  }
}

/*
 * This function allows a subscriber to subscribe to a topic by using
 data passed in from a subscription packet
 * and the subscriber's address.
 */
private boolean subscribe(byte[] data, SocketAddress subscriberAddress) {
  String nameOfTopic = ProtocolUtility.getMessage(data);
  if (topicsToSubscribersMap.containsKey(nameOfTopic)) {
    ArrayList<InetSocketAddress> subscriberList =
    topicsToSubscribersMap.get(nameOfTopic);
    subscriberList.add((InetSocketAddress) subscriberAddress);
    topicsToSubscribersMap.remove(nameOfTopic);
    topicsToSubscribersMap.put(nameOfTopic, subscriberList);
    terminal.println("A new subscriber has just subscribed to the
    topic:" + nameOfTopic + "!");
    return true;
```

```
    }
    return false;
  }

  /*
   * This function publishes a message for a topic by using data passed in
   from a publication packet.
   */
  private boolean publish(byte[] data) {
    int topicNumber = ProtocolUtility.getTopicNumber(data);
    ProtocolUtility.setType(data, PUBLISH);
    if (topicNumbersToTopicNamesMap.containsKey(topicNumber)) {
      String nameOfTopic = topicNumbersToTopicNamesMap.get(topicNumber);
      ArrayList<InetSocketAddress> dstAddressList =
      topicsToSubscribersMap.get(nameOfTopic);
      if (!dstAddressList.isEmpty()) {
        for (int i = 0; i < dstAddressList.size(); i++) {
          try {
            socket.send(new DatagramPacket(data, data.length,
            dstAddressList.get(i)));
            terminal.println("The topic " + nameOfTopic + " was published
            to!");
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
      return true;
    }
    return false;
  }
```

The Broker waits for any incoming packets from any other nodes, and from the header information it can determine which type of packet it is, and therefore complete the correct action such as creation of a topic, subscription to a topic, or publication to a topic, sending back an acknowledgment indicating whether the action was successful or not. If the packet is an acknowledgement (ACK packet), the Broker will output a statement saying it has been received. If the packet is a subscription (SUBSCRIBE), the subscriber's port number is added to the topicsToSubscribersMap if the topic exists. If the packet is a publication (PUBLISH) and the topic actually exists, the broker will send the message to all the subscribers via their port number. If the packet is a creation (CREATE_TOPIC), then the broker will add it to the appropriate HashMaps to update them if needed.

3.4 Subscriber.java

The subscriber takes in input (to choose what it wants to subscribe to) and output using the Terminal. The subscriber can only subscribe to one topic at a time and stop taking in any input after a subscription. This is because my implementation could not wait on packets as it did not print published messages as it was waiting on user input, requiring threading to implement.

```
/*
 * This function allows the subscriber to subscribe to a topic by taking
 user input about the name of the topic and sending a
 * subscription packet to the broker.
 */
public synchronized void subscribe() {
  String data = terminal.readString("Enter a topic to subscribe to!: ");
  terminal.println("Sending packet now!");
  try {
    socket.send(createPackets(SUBSCRIBE, 0, data, dstAddress)[0]);
  } catch (IOException e) {
  }
  terminal.println("Packet has been sent!");
}


/*
 * This function implements the subscriber's functionality - It can
 subscribe to a topic to receive communication from publishers.
 */
public synchronized void start() throws Exception {
  while (incorrectInput == true) {
    String startingString = terminal.readString("Enter SUBSCRIBE to
    subscribe to a topic: ");
    if (startingString.equals("SUBSCRIBE")) {
      subscribe();
      this.wait(); // wait for the ack first
      this.wait(); // then wait for the message
    } else {
      terminal.println("Invalid input.");
      incorrectInput = true;
    }
  }
  while (true) {
    this.wait();
  }
}
```

The first while loop in the start method runs while the Subscriber does not know if it's subscription is successful or not. The second while loop runs for an infinite amount of time, waiting for messages from the broker about the subscribed topic. I wnated to implement an unsubscribe function, although it was not necessary for this assignment but due to time constraints I was unable to. The subscribe function however sends subscription (SUBSCRIBE) packets to the broker if the user wishes to subscribe to a topic that exists. The onReceipt function checks if the packet sent to the Subscriber is an acknowledgement, publication, or message.

For this assignment, you need to make multiple Subscribers. I have included two Subscribers with my source code, but this can be extended to have more Subscribers. I have two files "Subscriber.java" and "Subscriber2.java" which have similar code, except they both have different port numbers. You could make another Subscriber if needed, but for

simplicity I called them Subscriber and Subscriber2, so that they can be used for multiple use cases such as for sensors or actuators.

3.5 Publisher.java

The Publisher takes input to create or join a topic. The publisher uses the HashMap in the ProtocolUtility class, and assigns a topic number to a topic when it's being created. This HashMap will be agreed with the Broker, and the topic number is sent as a creation packet (CREATE_TOPIC) to the Broker with the topic name. Therefore, the Publisher can input the topic as a String, represented as a integer of length 4 bytes. This allows it to be a fixed length in the publication packet (PUBLISH), with the payload/message at the end.

```
/*
   * This function publishes a message by taking user input and
   communicating with the broker to publish a message
   * to all the current subscribers of the current topic.
   */
  private boolean publishMessage() {
    String topic = terminal.readString("What's the name of the topic you
    want to publish a message for?: ");
    String message = terminal.readString("What's the message that you
    would like to publish?: ");
    int topicNumber = Integer.MAX_VALUE;
    for (int i = 0; i < ProtocolUtility.topicNumbersToNamesMap.size();
    i++) {
      if ((ProtocolUtility.topicNumbersToNamesMap.get(i)).equals(topic)) {
        topicNumber = i;
      }
    }
    if (topicNumber == Integer.MAX_VALUE) {
      terminal.println("I dont think this topic exists!");
    }
    else {
      DatagramPacket[] publicationPackets = createPackets(PUBLISH,
      topicNumber, message, dstAddress);
      try {
        terminal.println("Sending packet now!");
        socket.send(publicationPackets[0]);
      } catch (IOException e) {
        e.printStackTrace();
      }
      terminal.println("Packet has been sent!");
      return true;
    }
    return false;
  }

  /*
   * This function creates a topic by using user input for the topic name
   and sending a packet to the
   * broker to create the appropriate packet.
   */
  private void createTopic() {
```

```
    String topic = terminal.readString("Enter a topic to create/join: ");
    terminal.println("Sending packet now!");

    DatagramPacket[] creationPackets = createPackets(CREATE_TOPIC,
    ProtocolUtility.topicNumbersToNamesMap.size(), topic, dstAddress);
    ProtocolUtility.topicNumbersToNamesMap.put(ProtocolUtility.topicNumbers
    ToNamesMap.size(), topic);
    try {
      socket.send(creationPackets[0]);
    } catch (IOException e) {
      e.printStackTrace();
    }
    terminal.println("Packet has been sent!");
  }
```

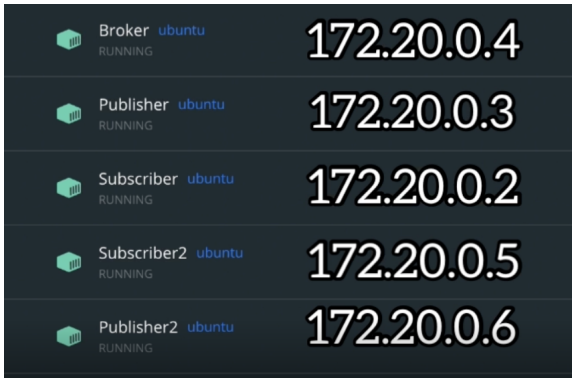
As seen above, the Publisher assigns a topic number when creating a topic, used later when making publication packets (PUBLISH). The Broker will get the topic number via the creation packet (CREATE_TOPIC) to figure out which topic the publish was requested for.

For this assignment, you need to make multiple Publishers. I have included two Publishers with my source code, but this can be extended to have more Publishers. I have two files "Publisher.java" and "Publisher2.java" which have similar code, except they both have different port numbers. You could make another Publisher if needed, but for simplicity I called them Publisher and Publisher2, so that they can be used for multiple use cases such as for sensors or actuators.

4 Discussion

4.1 Docker

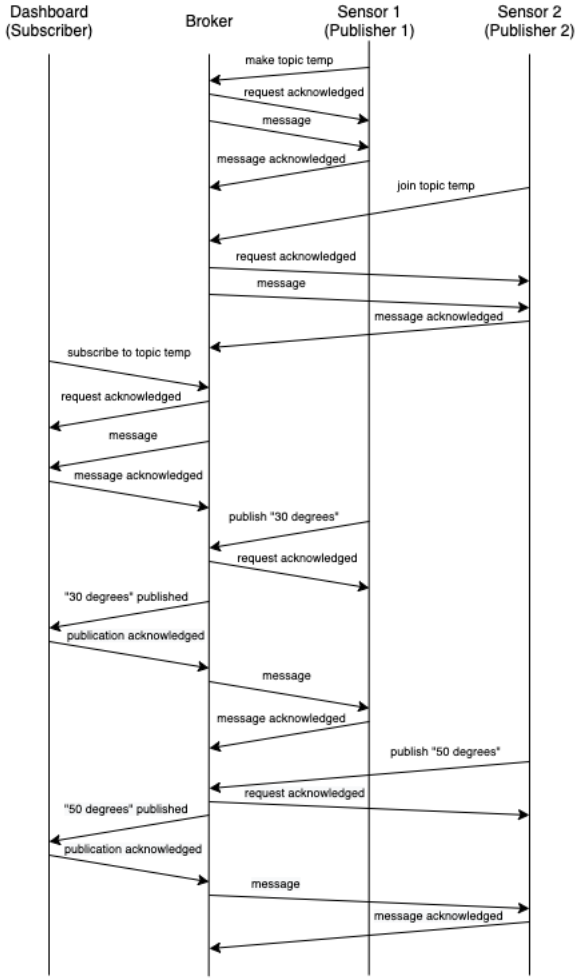
My protocol can connect components located at a number of hosts. This can be demonstrated with Docker, where I can run the Broker, Publisher(s) and Subscriber(s) inside of Docker containers, with each of them having their own IP Address. I had to use Ubuntu containers instead of Java containers as the java image for docker containers is only available for Intel processors. I was unaware of this at the start, and as a result it took me a while to get my implementation working on Docker

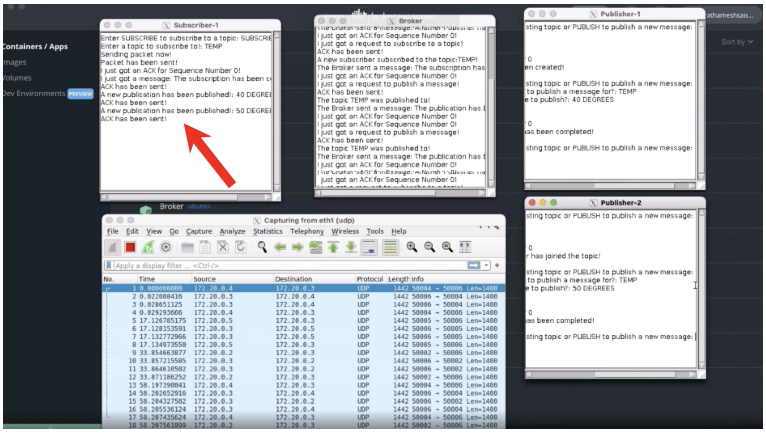


4.2 Communication between Nodes

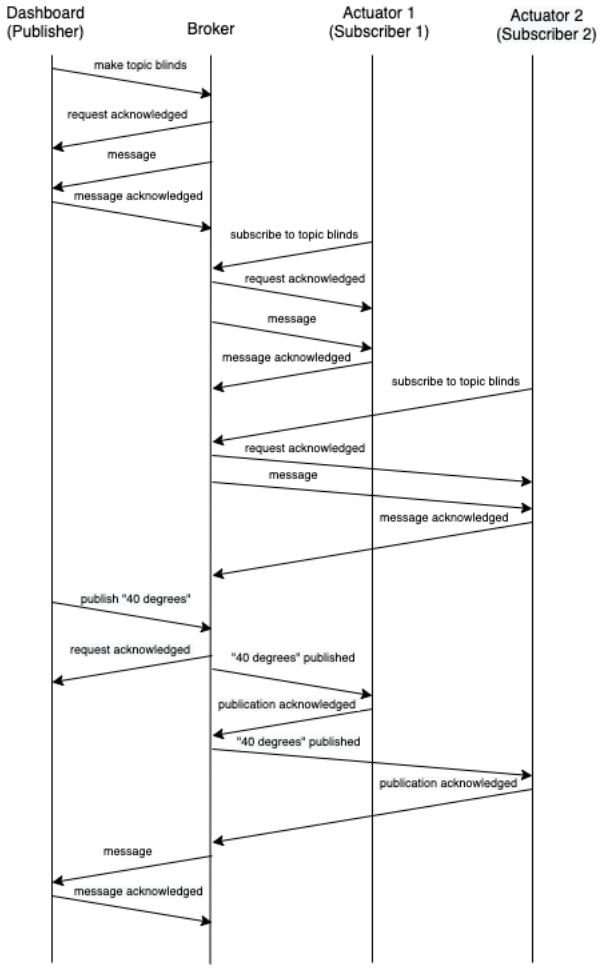
The Subscriber and Publisher are given ”Broker” as the default destination (hostname) along with the Broker’s port number to send all their packets. Therefore, there is no communication between the Subscriber and the Publisher directly. However the Broker has no destination port as it sends information to both the Subscriber and the Publisher. The broker does not know anything about the port numbers of a node until it is sent packets from them, when it can store them for later use if it is a subscription packet (SUBSCRIBE).

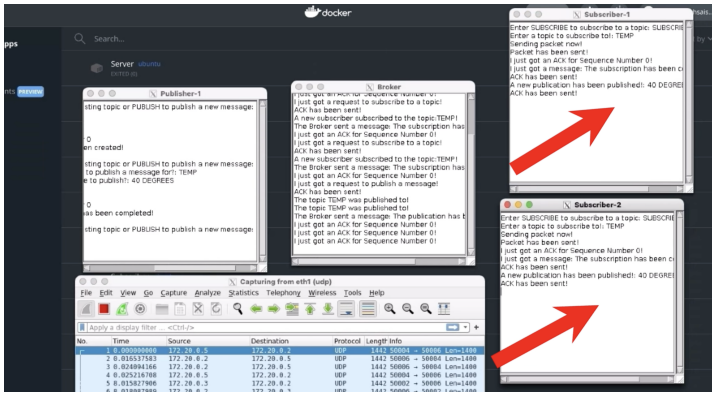
For the first use case in the assignment description, there is a Dashboard (Subscriber) with 2 Sensors (Publishers). My implementation would look something like this (you can follow along with the file 2Pub-1Sub-Prathamesh-Sai-19314123.pcap submitted with this assignment).





For the second use case in the assignment description, there is a Dashboard (Publisher) with 2 Actuators (Subscribers). My implementation would look something like this (you can follow along with the file 1Pub-2Sub-Prathamesh-Sai-19314123.pcap submitted with this assignment).





5 Summary

In this report I have described my implementation in Java of the Publish Subscribe protocol. My implementation used 5 classes. An abstract Node class, a utility class, a Broker class, a Subscriber class, and a Publisher class. The Broker, Subscriber and Publisher are extensions of the Node class, with unique functionality in the onReceipt function to react differently to incoming packets.

Node class: an abstract class that creates packets and their data, sends acknowledgments, and listens for packets.
ProtocolUtility class: A utility class providing utility functions to be used by other classes to avoid the repetition of code. Also contains a HashMap to keep track of the topic numbers.
Publisher: Creates topics, publishes messages to topics, and handles user input/output.
Subscriber: Subscribes to topics, waits for publishers to publish messages, and handles user input/output.

6 Reflection

As I reflect on the process of implementing this protocol, I thoroughly enjoyed it. I learnt how to design the layout of a packet, how to implement simple acknowledgments, and reinforce the information I learnt during the lectures. I had many challenges to face such as being new to Docker and finding threading very difficult to make Subscribers concurrently accept packets and subscribe to multiple topics. I found very simple commands such as copying my code to my docker container using docker cp difficult as I had never used it before. Finally, I found it difficult to get a Node to accept a message split into multiple packets. These problems took me weeks to overcome after building my initial basic solution for video 1, which took a lot of time away from implenting other features such as error control and flow control. However, I think I learnt a lot during the process of building my final implementation, even with the challenges I faced. I hope to use the lessons I learnt during this assignment in the second assignment.