



---

# CS2031 Telecommunications II

## Assignment #1: Publish-Subscribe

---

Jack Gilbride, Student ID #17340868

December 2, 2018

### Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theory of Topic</b>	<b>2</b>
2.1	Internet Protocol . . . . .	2
2.2	User Datagram Protocol . . . . .	2
2.3	Maximum Transmission Unit . . . . .	2
<b>3</b>	<b>Implementation</b>	<b>2</b>
3.1	Node . . . . .	2
3.2	Acknowledgements . . . . .	5
3.3	Broker . . . . .	5
3.4	Subscriber . . . . .	8
3.5	Publisher . . . . .	10
<b>4</b>	<b>Discussion</b>	<b>12</b>
4.1	Use of IP Addresses and Port Numbers . . . . .	12
4.2	Custom Packet Layout . . . . .	13
<b>5</b>	<b>Summary</b>	<b>13</b>
<b>6</b>	<b>Reflection</b>	<b>13</b>

## 1 Introduction

The task of this assignment was to design a Publish-Subscribe protocol. A Publish-Subscribe protocol is a protocol that forwards messages from a Publisher to a Broker, which in turn distributes these messages to a number of Subscribers. In this report I will describe my approach towards this assignment, starting with the theory of the topic, then describing my own implementation of the protocol in Java. I will finish my report with a discussion, a summary and a reflection on the overall assignment.

## 2 Theory of Topic

In this section I will describe the concepts and protocols that were used to realise a solution to my final Publish-Subscribe protocol. In particular I will describe Internet Protocol and User Datagram Protocol. These are necessary to understand as they were used by the `java.net.DatagramPacket` class, which was integral to the communication between the Nodes in the protocol. I will also briefly describe the meaning of a Maximum Transmission Unit.

### 2.1 Internet Protocol

Internet Protocol (IP) is a protocol in the Network Layer of the Open Systems Interconnection (OSI) model. It is the only protocol used in the Network Layer of the Internet OSI model. Two versions of IP currently coexist in the global Internet; IP version 4 (IPv4) and IP version 6 (IPv6). As the task of the Network Layer is to move packets from source to destination, an integral piece of information in an IP header is the IP address. In IPv4 the IP address is 32 bits (4 bytes) long, whereas in IPv6 it is 128 bits (16 bytes) long. In both versions, the IP header will contain the IP address of the source host (source address) and the IP address of the destination host (destination address). Routers in the Internet use these IP addresses to forward packets to the correct destination. In the `java.net` library, an IP Address is represented by an `InetAddress` object, which may use either IPv4 or IPv6. IP Addresses are the only information needed to understand the role of IP in the `DatagramPacket`.

### 2.2 User Datagram Protocol

User Datagram Protocol is a protocol in the Transport Layer of the OSI model. It is used in the Internet OSI model where it is built on top of IP. A UDP header is only 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 its calculation and inclusion are optional in UDP. Because of its short length and small amount of fields, UDP is a connectionless, unreliable protocol. It has no flow control and optional error control, which makes it useful for applications that provide their own flow and error control. Port numbers are the most important piece of information in UDP to understand for this assignment as every node needs a different port number to communicate effectively in this protocol. However it is also important to know that UDP has a small header so most of the information used in the `DatagramPacket` will come from higher levels of the OSI model, which can be implemented efficiently by the Nodes in the protocol.

### 2.3 Maximum Transmission Unit

The maximum size of a frame that can be sent depends on underlying hardware architecture. The maximum frame size of the medium determines its Maximum Transmission Unit (MTU). For example, the MTU for the Ethernet protocol is 1,500, so a frame of size up to 1,500 bytes can be sent with this protocol. It is common for information to be split up into different packets on the Internet to accommodate the MTU of the medium, a process known as fragmentation. Fragmentation was not used in this assignment but it was still important to determine a maximum packet size for the protocol so that it could accommodate relatively small MTUs.

## 3 Implementation

### 3.1 Node

The Broker, Publisher and Subscriber were all extensions of the abstract Node class that we were provided on 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. The code in listing 1 shows the entire functionality of the Listener class.

```
class Listener extends Thread {  
    /*  
     * Telling 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();  
            // Endless loop: attempt to receive packet, notify receivers, etc  
            while(true) {  
                DatagramPacket packet = new DatagramPacket(new byte[PACKETSIZE],  
                    PACKETSIZE);  
                socket.receive(packet);  
                onReceipt(packet);  
            }  
        } catch (Exception e) {if (!(e instanceof SocketException))  
            e.printStackTrace();}  
    }  
}
```

Listing 1: The Listener Class. Note how the received packet is handled by onReceipt, which must be implemented in its own way by the Broker, Publisher and Subscriber.

Any functionality or constants needed by all three subclasses was also added into Node. PACKETSIZE was set to 1400, which meant that the length of any packet could not exceed 1400 bytes. This was done so that the system could work on media with relatively small MTUs. DEFAULT\_DST was the name of the IP address for all three Nodes, chosen to be localhost. This meant that all three Nodes ran on the same machine. The three Nodes were all individual processes so were given individual port numbers. Due to time constraints, the protocol did not extend beyond one of each type of Node so the IP Address and port numbers remained as constants in the final implementation. Other constants were added for packet types; ACK, NAK, RTS, CTS, CREATION, PUBLICATION, SUBSCRIPTION, UNSUBSCRIPTION and MESSAGE. Also due to time constraints, negative acknowledgements (NAK), request to send (RTS) and clear to send (CTS) were not implemented as functionality in the final system.

The packets used in this protocol were Datagram Packets. The UDP header would contain the source and destination port numbers. The IP header contained the localhost IP Address as both the source and destination IP Addresses. It was up to us to implement the upper layers of the protocol, contained in the array of bytes in the DatagramPacket. A fixed-length packet was implemented. Byte 0 was the type of packet, one of the nine constants named above. Byte 1 was the sequence number of the packet. This was put into the protocol with the intention of implementing error control such as Go-Back-N. However due to time constraints this was never implemented, so byte 1 is always set to the value 0 in the final system. Bytes 2, 3, 4 and 5 contained the topic number, i.e. the agreed integer corresponding to a topic in the system. This is described in greater detail in the explanation of Publisher. The remaining bytes contained a string of characters with a message such as the topic to subscribe to or the message to publish. The code in listing 2 shows how the node creates packets.

```
/* Take the type of packet, topic number, message and destination address and  
 * return an array of one or more packets. Assumes a message can be stored in  
 * multiple packets, but the actual program will only work if the message fits  
 * in one packet.
```

```

*/
protected DatagramPacket[] createPackets(int type, int topicNumber,
    String message, InetAddress dstAddress){
    int messageSize = PACKETSIZE-6;
    byte[] tmpArray = message.getBytes();
    byte[] messageArray = new byte[tmpArray.length];
    for(int i=0; i<tmpArray.length; i++){
        messageArray[i]=tmpArray[i];
    }
    int numberOfPackets=0;
    for(int messageLength = messageArray.length; messageLength>0;
        messageLength-=messageSize){
        numberOfPackets++;
    }
    DatagramPacket[] packets = new DatagramPacket[numberOfPackets];
    int offset = 0;
    for(int sequenceNumber=0; sequenceNumber<numberOfPackets; sequenceNumber++){
        byte[] dividedMessage = new byte[messageSize];
        for(int j=offset; j<offset+messageArray.length; j++){
            dividedMessage[j]=messageArray[j+offset];
        }
        byte[] data = createPacketData(type, sequenceNumber, topicNumber,
            dividedMessage);
        DatagramPacket packet = new DatagramPacket(data, data.length, dstAddress);
        packets[sequenceNumber]=packet;
        offset+=messageSize;
    }
    return packets;
}

/* Create an array of bytes for a DatagramPacket and returns it. Based on
 * custom packet data layout; byte 0 = type, byte 1 = sequence number for
 * Go-Back-N, bytes 2-5 = topic number, remaining bytes = message.
 */
private byte[] createPacketData(int type, int sequenceNumber, int topicNumber,
    byte[] message){
    byte[] data = new byte[PACKETSIZE];
    data[0]=(byte)type; // Set type
    data[1]=(byte)sequenceNumber;
    ByteBuffer byteBuffer = ByteBuffer.allocate(4);
    byte[] topicNumberArray = byteBuffer.array();
    for(int i=0; i<4; i++){
        data[i+2] = topicNumberArray[i];
    }
    for(int i=0; i<message.length && i<PACKETSIZE; i++){
        data[i+6]=message[i];
    }
    return data;
}

```

Listing 2: Creating an array of packets given the type of packets, topic number, message and destination address. The methods in the listing work for message lengths longer than can fit in one packet, however the rest of the program cannot implement the sending or receiving of multiple packets correctly.

The remainder of the node class contains methods to get the type, sequence number, topic number and

message of a byte array, along with a method to send an acknowledgement, which would be implemented by all three types of node.

### 3.2 Acknowledgements

Simple acknowledgements were implemented in this program. Whenever the Broker, Subscriber or Publisher sent a packet it would wait for an acknowledgement. Publishers and Subscribers would also wait for a message on whether their publication, creation, subscription or unsubscription was unsuccessful. These were implemented separately because acknowledgements were purely to inform the Node that their packet was received, whereas messages were purely for the benefit of the person running the program. The method to send acknowledgements was in the Node class and can be seen in listing 3.

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

protected void sendAck(DatagramPacket receivedPacket, Terminal terminal) {
    byte[] data = receivedPacket.getData();
    setType(data, ACK);
    DatagramPacket confirmation = new DatagramPacket(data, data.length,
        receivedPacket.getSocketAddress());
    try {
        socket.send(confirmation);
        terminal.println("Sent ACK.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Listing 3: Sending an acknowledgement. This is a simple method which takes the received packet and sends it back to the original sender, with the type set to ACK. The receiver of an ACK only cares about the type of packet so will not check the message field.

### 3.3 Broker

The Broker contains three global classes; a terminal and two HashMaps. The terminal is purely for output; no input is taken for the Broker as it is purely an interface between the Publisher and Subscriber. The first map, called subscriberMap, maps the names of topics to the list of port numbers of its Subscribers. Although there is only one Subscriber in the current system, the decision to use a list was made early in the design process with multiple Subscribers in mind and could be easily extended to work with multiple Subscribers. The second map is called topicNumbers, which contains the topic numbers as keys and topic names as values. Having a topic number for each topic name means that a Publisher can easily publish a message using the fixed packet layout, with the topic number in bytes 2-5 and the message in the remaining bytes. This is explained in greater detail in the 'Publisher' subsection.

When the Broker starts, it waits for incoming packets. The Broker does not know the port numbers or IP Addresses of other Nodes in the system so it waits for contact from them. Inside the onReceipt method is a switch statement which switches based on the type of packet received. In the case of an ACK, the Broker will print to the terminal that it was received. In the case of a subscription, unsubscription, creation or publication, it will send back an ACK, call the relevant method and reply to the sender, indicating whether the operation was successful or unsuccessful. The code in listing 4 shows the functions called when the packet is a creation or a publication, and the code in listing 5 shows those called for a subscription or an unsubscription.

```

private boolean createTopic(byte[] data) {
    ArrayList<InetSocketAddress> socketNumbers =
        new ArrayList<InetSocketAddress>();
    String topicName = getMessage(data);
    if (!subscriberMap.containsKey(topicName)) {
        subscriberMap.put(topicName, socketNumbers);
        int topicNumber = getTopicNumber(data);
        topicNumbers.put(topicNumber, topicName);
        terminal.println("Topic_" + topicName + "_was_created.");
        return true;
    }
    return false;
}

private boolean publish(byte[] data) {
    int topicNumber = getTopicNumber(data);
    setType(data, PUBLICATION);
    if (topicNumbers.containsKey(topicNumber)) {
        String topicName = topicNumbers.get(topicNumber);
        ArrayList<InetSocketAddress> dstAddresses = subscriberMap.get(topicName);
        if (!dstAddresses.isEmpty()) {
            for (int i = 0; i < dstAddresses.size(); i++) {
                DatagramPacket publication = new DatagramPacket(data, data.length,
                    dstAddresses.get(i));
                try {
                    socket.send(publication);
                    terminal.println("Topic_" + topicName + "_was_published.");
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return true;
    }
    return false;
}

```

Listing 4: The Broker's interaction with the Publisher. If the received packet is a creation, the Broker will add it to the relevant maps if it is not already there. If it is a publication and the topic exists, the Broker will forward the message to the relevant Subscriber port numbers.

```

private boolean subscribe(byte[] data, SocketAddress subscriberAddress) {
    String topicName = getMessage(data);
    if (subscriberMap.containsKey(topicName)) {
        ArrayList<InetSocketAddress> subscribers = subscriberMap.get(topicName);
        subscribers.add((InetSocketAddress) subscriberAddress);
        subscriberMap.remove(topicName);
        subscriberMap.put(topicName, subscribers);
        terminal.println("A_new_subscriber_subscribed_to_" + topicName + ".");
        return true;
    }
    return false;
}

```

```
private boolean unsubscribe(byte[] data, SocketAddress subscriberAddress) {
    boolean unsubscribed = false;
    String topicName = getMessage(data);
    if (subscriberMap.containsKey(topicName)) {
        ArrayList<InetSocketAddress> subscribers = subscriberMap.get(topicName);
        if (!subscribers.isEmpty()) {
            for (int i = 0; i < subscribers.size(); i++) {
                if (subscribers.get(i).equals(subscriberAddress)) {
                    subscribers.remove(i);
                    terminal.println("A subscriber unsubscribed from " + topicName + ".");
                    unsubscribed = true;
                }
            }
        }
        subscriberMap.remove(topicName);
        subscriberMap.put(topicName, subscribers);
    }
    return unsubscribed;
}
```

Listing 5: The Broker's interaction with the Subscriber. If the packet is a subscription and the topic exists, then the subscriber's port number is added to the subscriberMap. If it is an unsubscription and the topic exists, the port number will be removed.

When handling acknowledgements, publications, creations, subscriptions and unsubscriptions, the Broker will print relevant information to its terminal about what it is doing. This terminal was an object of a class given to us on Blackboard. This terminal class was also used by the Publisher and the Subscriber. See figure 1 to view an example of the Broker's terminal.

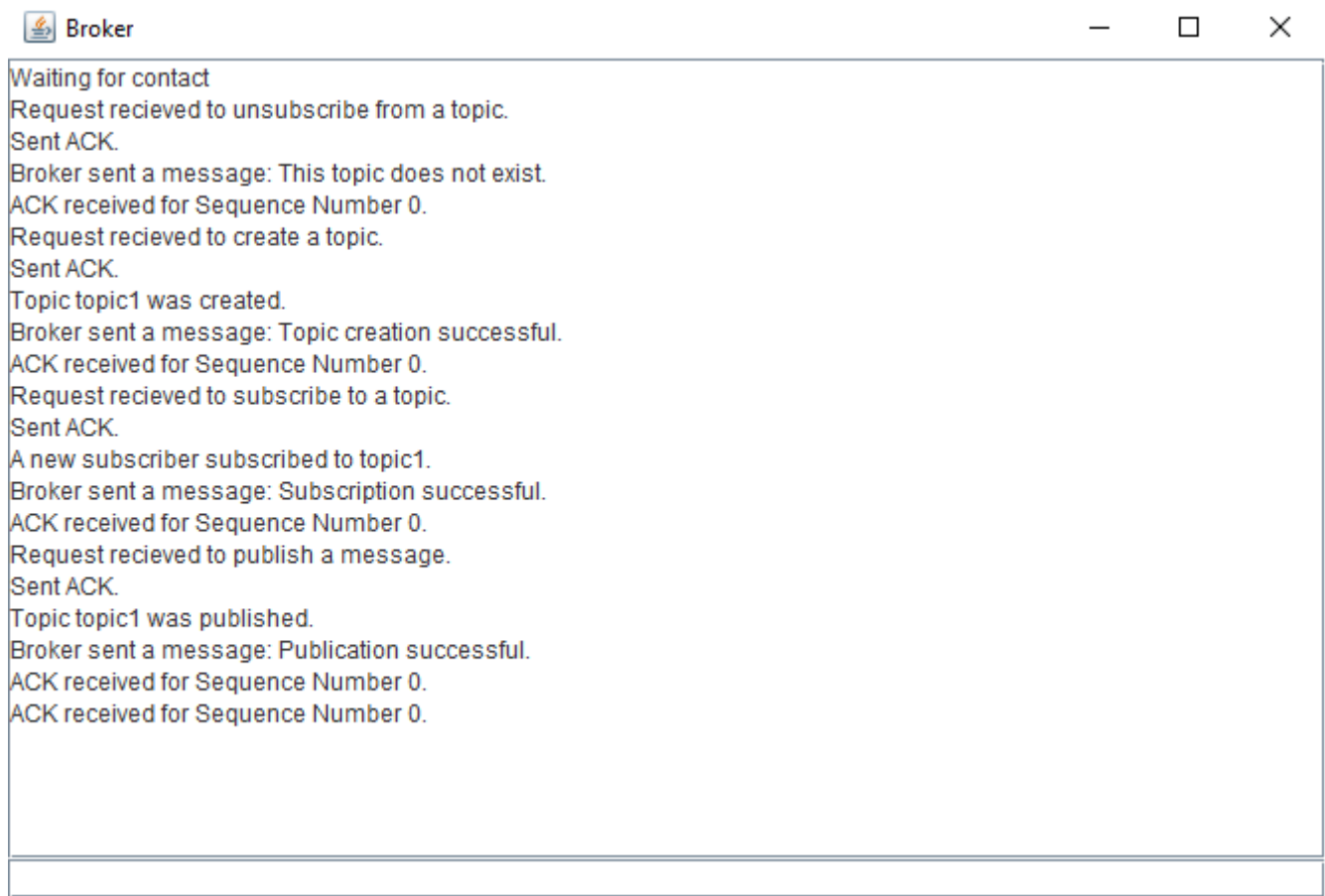


Figure 1: Terminal of a Broker after interactions with a Publisher and a Subscriber. Pictured; an attempt to unsubscribe from a nonexistant topic, creation of a topic, subscription to the new topic and publication of a message for the new topic.

### 3.4 Subscriber

The Subscriber has a global Terminal, a destination address and a boolean called `invalidInput`. The terminal is used for both input and output so that the user can decide what it wants the Subscriber to do. The original implementation of the Subscriber was to be able to subscribe and unsubscribe from as many topics as possible in any order. However towards the end of my design I realised that my implementation did not print any published messages because it was busy waiting on user input, therefore could not wait on packets. Solving this problem would have involved threading which I found very complicated to properly implement and decided not to do due to time constraints. As a result, the Subscriber will stop taking input once it has successfully subscribed to a topic. This implementation allows for the majority of the functionality of the Subscriber to be shown while also allowing it to receive and print messages. Snippet 6 shows the `start()` method of the Subscriber, which is run upon startup.

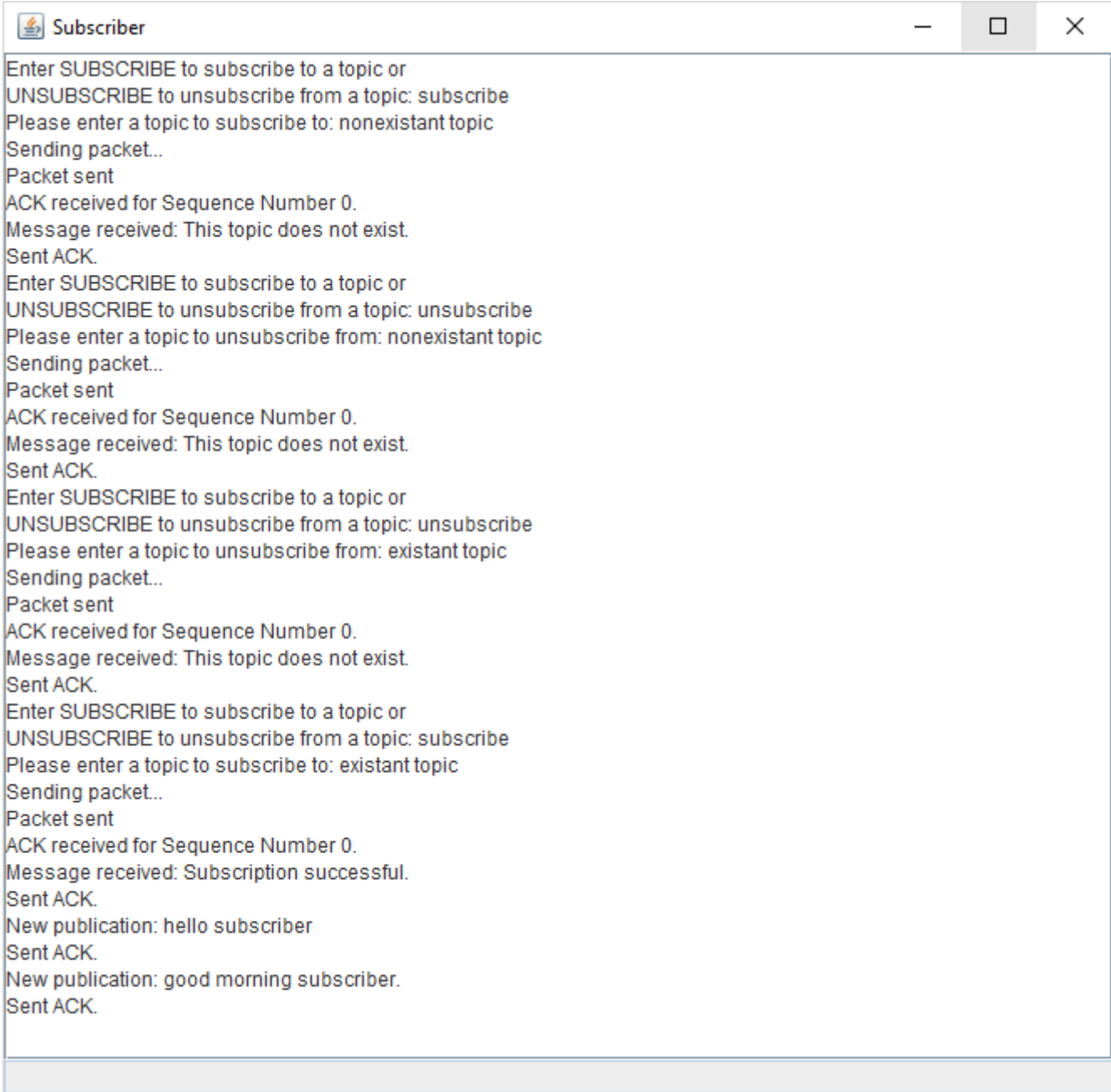
```
public synchronized void start() throws Exception {
    while (invalidInput == true) {
        String startingString = terminal.read(
            "Enter SUBSCRIBE to subscribe to a topic or UNSUBSCRIBE to unsubscribe"
            + "from a topic: ");
        terminal.println("Enter SUBSCRIBE to subscribe to a topic or"
            + "\nUNSUBSCRIBE to unsubscribe from a topic: " + startingString);
        if (startingString.toUpperCase().contains(UNSUBSCRIBE)) {
```



```
unsubscribe();
this.wait(); // wait for ACK
this.wait(); // wait for MESSAGE
} else if (startingString.toUpperCase().contains(SUBSCRIBE)) {
    subscribe();
    this.wait(); // wait for ACK
    this.wait(); // wait for MESSAGE
} else {
    terminal.println("Invalid input.");
    invalidInput = true;
}
}
while (true){
    this.wait();
}
}
```

Listing 6: The main functionality of Subscriber. The first loop will run as long as the Subscriber has not been told by the Broker that its subscription is successful. After that the second loop will run infinitely, waiting for messages about the topic that it has subscribed to.

The subscribe() and unsubscribe() functions ask the user which topic they would like to subscribe to/unsubscribe from and sends the relevant packets to the broker. The onReceipt() function checks whether the incoming packet is an ACK, message or publication and prints the relevant information. While in the second loop, i.e. the second state, of start(), the Subscriber will only print incoming messages and publications. Figure 2 shows a sample terminal output of Subscriber, showing the different functionality that it is capable of.



```
Subscriber
Enter SUBSCRIBE to subscribe to a topic or
UNSUBSCRIBE to unsubscribe from a topic: subscribe
Please enter a topic to subscribe to: nonexistent topic
Sending packet...
Packet sent
ACK received for Sequence Number 0.
Message received: This topic does not exist.
Sent ACK.
Enter SUBSCRIBE to subscribe to a topic or
UNSUBSCRIBE to unsubscribe from a topic: unsubscribe
Please enter a topic to unsubscribe from: nonexistent topic
Sending packet...
Packet sent
ACK received for Sequence Number 0.
Message received: This topic does not exist.
Sent ACK.
Enter SUBSCRIBE to subscribe to a topic or
UNSUBSCRIBE to unsubscribe from a topic: unsubscribe
Please enter a topic to unsubscribe from: existant topic
Sending packet...
Packet sent
ACK received for Sequence Number 0.
Message received: This topic does not exist.
Sent ACK.
Enter SUBSCRIBE to subscribe to a topic or
UNSUBSCRIBE to unsubscribe from a topic: subscribe
Please enter a topic to subscribe to: existant topic
Sending packet...
Packet sent
ACK received for Sequence Number 0.
Message received: Subscription successful.
Sent ACK.
New publication: hello subscriber
Sent ACK.
New publication: good morning subscriber.
Sent ACK.
```

Figure 2: The terminal of the Subscriber after interaction with the Broker. Pictured; Subscription to a nonexistent topic, unsubscription from a nonexistent topic, unsubscription from a topic that it is not subscribed to, successful subscription, receiving publications.

### 3.5 Publisher

The Publisher is similar to the Subscriber and interacts solely with the Broker. Instead of subscribing or unsubscribing from topics, it creates and publishes for them. As the Publisher does not have to deal with unexpected incoming packets, it may always wait for user input and, as a result, may create and publish for as many topics as needed. Like the Broker, the Publisher has a `HashMap` called `topicNumbers`, with topic numbers as keys and topic names as values. Ideally the Publisher and Broker will have identical copies of `topicNumbers`. When creating a topic, the publisher will assign a topic number to the topic and remember

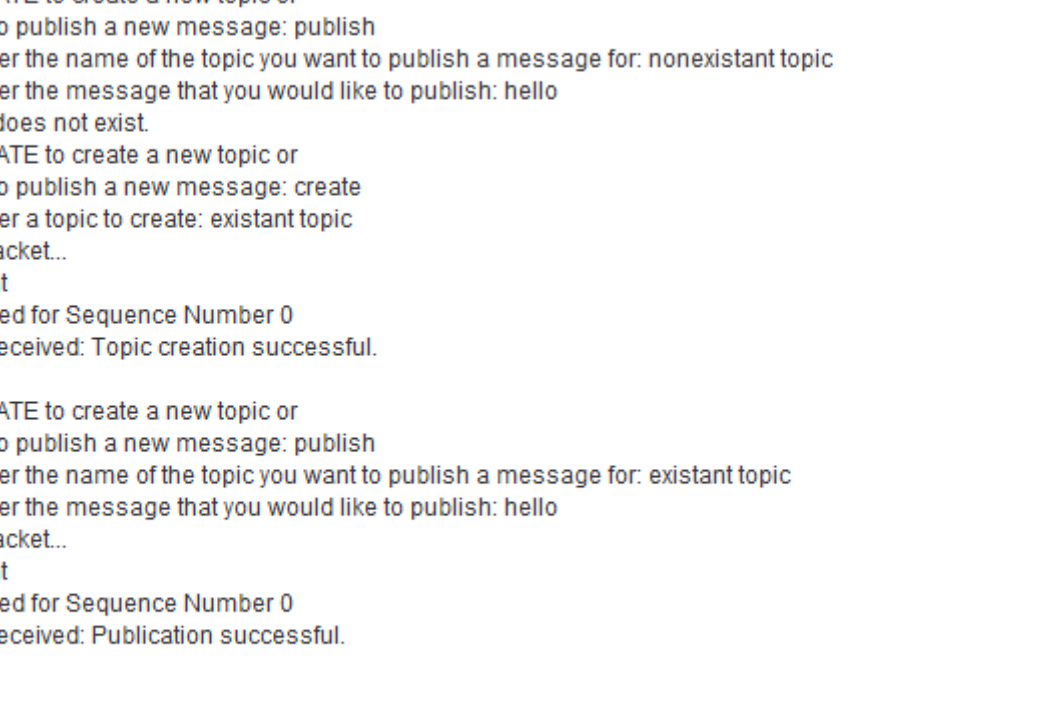
it. This topic number will be sent in a creation packet to the Broker, along with the topic name. This way the user controlling Publisher may later input the topic as a String, which can be represented as a 4-byte fixed-length integer within a publication packet. This means that the topic name can always be represented in a fixed length in a publication packet, with the message taking up the remainder of the packet. The topic numbers start at zero and can go up to  $2^{31} - 1$ . The way that the Publisher assigns and uses these topic numbers can be seen in the methods in snippet 7.

```
private void createTopic() {
    String topic = terminal.read("Please_enter_a_topic_to_create:");
    terminal.println("Please_enter_a_topic_to_create:" + topic);
    terminal.println("Sending_packet...");
    DatagramPacket[] packets = createPackets(CREATION, topicNumbers.size(), topic,
        dstAddress);
    topicNumbers.put(topicNumbers.size(), topic);
    try {
        socket.send(packets[0]);
    } catch (IOException e) {
        e.printStackTrace();
    }
    terminal.println("Packet_sent");
}

private boolean publishMessage() {
    String topic = terminal.read("Please_enter_the_name_of_the_topic_you_want_to_"
        + "publish_a_message_for:");
    terminal.println("Please_enter_the_name_of_the_topic_you_want_to_publish_"
        + "a_message_for:" + topic);
    String message = terminal.read("Please_enter_the_message_that_you_would_like_"
        + "to_publish:");
    terminal.println("Please_enter_the_message_that_you_would_like_to_publish:"
        + message);
    int topicNumber = Integer.MAX_VALUE;
    for (int i = 0; i < topicNumbers.size(); i++) {
        if ((topicNumbers.get(i)).equals(topic)) {
            topicNumber = i;
        }
    }
    if (topicNumber == Integer.MAX_VALUE) {
        terminal.println("This_topic_does_not_exist.");
    } else {
        DatagramPacket[] packets = createPackets(PUBLICATION, topicNumber, message,
            dstAddress);
        try {
            terminal.println("Sending_packet...");
            socket.send(packets[0]);
        } catch (IOException e) {
            e.printStackTrace();
        }
        terminal.println("Packet_sent");
        return true;
    }
    return false;
}
```

Listing 7: The methods used to send packets to the Broker. These are called after the user is asked for input in `start()`. When creating a topic, the Publisher assigns it a topic number which it can later use in corresponding publication packets. The Broker receives the topic number through the creation packet so can later find out which topic the publication is for.

Aside from the code in listing 7, the Publisher contains a start function, which infinitely gets input and calls the above methods, and an onReceipt function, which prints any received acknowledgements or messages. An example of interaction with the Publisher can be seen in figure 3.



**Publisher**

Enter CREATE to create a new topic or  
PUBLISH to publish a new message: publish  
Please enter the name of the topic you want to publish a message for: nonexistent topic  
Please enter the message that you would like to publish: hello  
This topic does not exist.  
Enter CREATE to create a new topic or  
PUBLISH to publish a new message: create  
Please enter a topic to create: existant topic  
Sending packet...  
Packet sent  
ACK received for Sequence Number 0  
Message received: Topic creation successful.  
Sent ACK.  
Enter CREATE to create a new topic or  
PUBLISH to publish a new message: publish  
Please enter the name of the topic you want to publish a message for: existant topic  
Please enter the message that you would like to publish: hello  
Sending packet...  
Packet sent  
ACK received for Sequence Number 0  
Message received: Publication successful.  
Sent ACK.

Enter CREATE to create a new topic or PUBLISH to publish a new message:

Figure 3: Terminal of a Publisher after interaction with a Broker. Pictured; trying to publish for a nonexistent topic, creating a topic, publishing for the created topic.

## 4 Discussion

In this section I will discuss some of the aspects of my protocol and how they make use of the material studied in lectures. In particular I will discuss the use of IP Addresses, use of port numbers and use of a custom packet layout in my implementation.

#### 4.1 Use of IP Addresses and Port Numbers

The current implementation of my protocol is essentially a simple model of a Publish-Subscribe protocol. Each node is on the same IP address so they must have different port numbers to indicate that they are different processes. Each Node has its own socket with a unique port number. The Publisher and Subscriber

are given the Broker's port number as a destination address for all packets. This way the Publisher and Subscriber have no direct contact, which adheres to the idea of the protocol. The Broker is given no set destination port as it must communicate with multiple nodes. It can only learn the port numbers of other nodes when it is contacted by them, and may store them for later use in the case of a subscription. Any replies or messages are sent immediately back from the Broker who sets the destination port number to the source port number of the packet that it has just received. The Broker can also tell what type of Node is contacting it; if the packet is a subscription or unsubscription then it is a Subscriber, and if the packet is a creation or publication then it is a Publisher. So overall the Nodes can still be viewed as separate and distinct processes because, although they are on the same IP Address, their port numbers keep them distinct.

## 4.2 Custom Packet Layout

The packet layout is an integral part of this protocol also. The custom packet layout is inside a datagram packet, following an IP header and a UDP header. Having a fixed index for each field of the data meant that they could be easily accessed using array notation. The sequence number was not checked by any of the nodes on receipt but it is a good example that packets may contain fields that are not always relevant. To conform to the protocol however, they may have to be included. The idea for my protocol was thought of before the final implementation, so the implementation must adhere to the protocol by having a sequence number. In this case the sequence number was always set to 0. An example of this in the real world is the "Options" field of an IPv4 header, which is seldom used but must still be included as a field.

The decision to include topic numbers as a way of referencing topics was also directly influenced by the fixed packet layout. I realised that the Broker and the Publisher could both have identical copies of topic numbers by allocating them using a creation packet. This made publication packets easy to create and parse due to the fixed length of the topic and address fields.

## 5 Summary

This report has described my attempt to create a simple Publish-Subscribe Protocol using Java. Overall my implementation consisted of a Terminal class from Blackboard, an abstract Node class, a Broker Node, a Subscriber Node and a Publisher Node. The three concrete Node classes all inherit from the abstract Node class, which provides common functionality to all three. The features of the overall system can be summarised as the following:

**Abstract Node Class:** Defining my Publish-Subscribe protocol layout, creating packets, listening for packets, sending acknowledgements.

**Broker:** Getting packet types, keeping track of topic numbers, handling subscriptions, handling unsubscriptions, handling topic creation, handling and forwarding publications, sending confirmation messages, printing output to user.

**Subscriber:** Handling user input, attempting to subscribe, attempting to unsubscribe, waiting for publications, printing output to user.

**Publisher:** Handling user input, keeping track of topic numbers, attempting to publish, attempting to create a topic, printing output to user.

## 6 Reflection

Upon reflection, I found the content of the assignment to be quite interesting. The use of DatagramPackets taught me about the Network and Transport Layers of the OSI model. I also learnt a lot about other features that were implemented in my Publish/Subscribe protocol, such as how to design a packet layout and how to implement simple acknowledgements. Overall, the research done for both the Java implementation and the report helped to reinforce the information that I learnt in lectures.

Unfortunately I ran into many problems during the assignment which were mainly to do with understanding how certain features should be implemented in Java. In particular I found threading very complicated and, as a result, could not separate the Subscriber into threads to concurrently accept packets and user input.

Another example is that I could not figure out how to get a Node to accept a message that had been split into multiple packets. Problems like these took me weeks to try to overcome, which took away from a lot of the time I spent on the assignment. As a result I could not implement many of the features that I wanted, such as better flow control and error control in the protocol. Overall I feel that I learnt a lot about Telecommunications from the assignment but my lack of knowledge in the Java implementation held back my final solution.

Despite my difficulties, this assignment also helped to teach me lessons that I could bring forward into Assignment 2. Assignment 2 was about OpenFlow so also involved communication between Nodes. Switches would be used to forward data between End Nodes, similar to the functionality of a Broker. I would continue to define my own packet layouts and parse them using arrays. I would also make sure that every Node and thread in Assignment 2 would start from the same mainline, as it would be much easier to start the entire system and manage how threads interact with one another. My lessons and struggles in Assignment 1 meant that I could quickly make up for lost time and implement a solution that I was satisfied with for Assignment 2.