
CS2031 Assignment #1

Publish & Subscribe

Kamil Przepiórowski

17327895

CONTENTS

1. Minimal implementation	3
2. My approach.	4
2.1 Design decisions.	5
3. Packet	7
3.1 Header design	7
4. Publisher	8
4.1 Syntax	8
4.2 Implementation	9
4.3 Stop-and-Wait	9
5. Broker.	11
5.1 Overall design	11
5.2 Relationship with publishers	12
5.3 Relationship with subscribers	12
5.4 Sample input and output	13
6. Subscriber	14
6.1 Syntax	14
6.2 Implementation	15
6.3 Subscribe	15
6.4 Unsubscribe	16
6.5 List topics	16
7. Reflection	17

1 Minimal Implementation

The first implementation of my program was minimal and consisted of just one Client (sender) and one Server (receiver), it was essentially a basic Stop-and-Wait program. The Client would send a packet with a message input by the user, and the Server would respond with an acknowledgment that it received the packet.

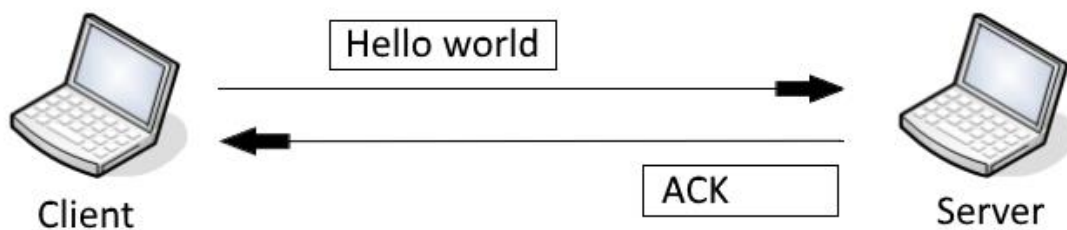


Figure 1: Simplistic exchange of packets between two nodes.

This would later evolve into the relationship between the publishers and the broker.

Having a minimal implementation like this allowed me to become familiar with using datagram packets and port addresses in Java, something which I haven't done before.

Once I familiarised myself with the syntax and how it all worked, it was time to implement the program on a larger scale. This required a lot of design decisions which affected how my system worked. I will discuss these in the following section.

2 My approach

The approach I took when designing the system put the **publishers** in charge of which topics were created, not the subscribers. This means that a **subscriber cannot** subscribe to a topic which does not exist yet. However, a **publisher can** publish to a topic which hasn't been created before; the broker will create a new topic in its HashMap if this is the case.

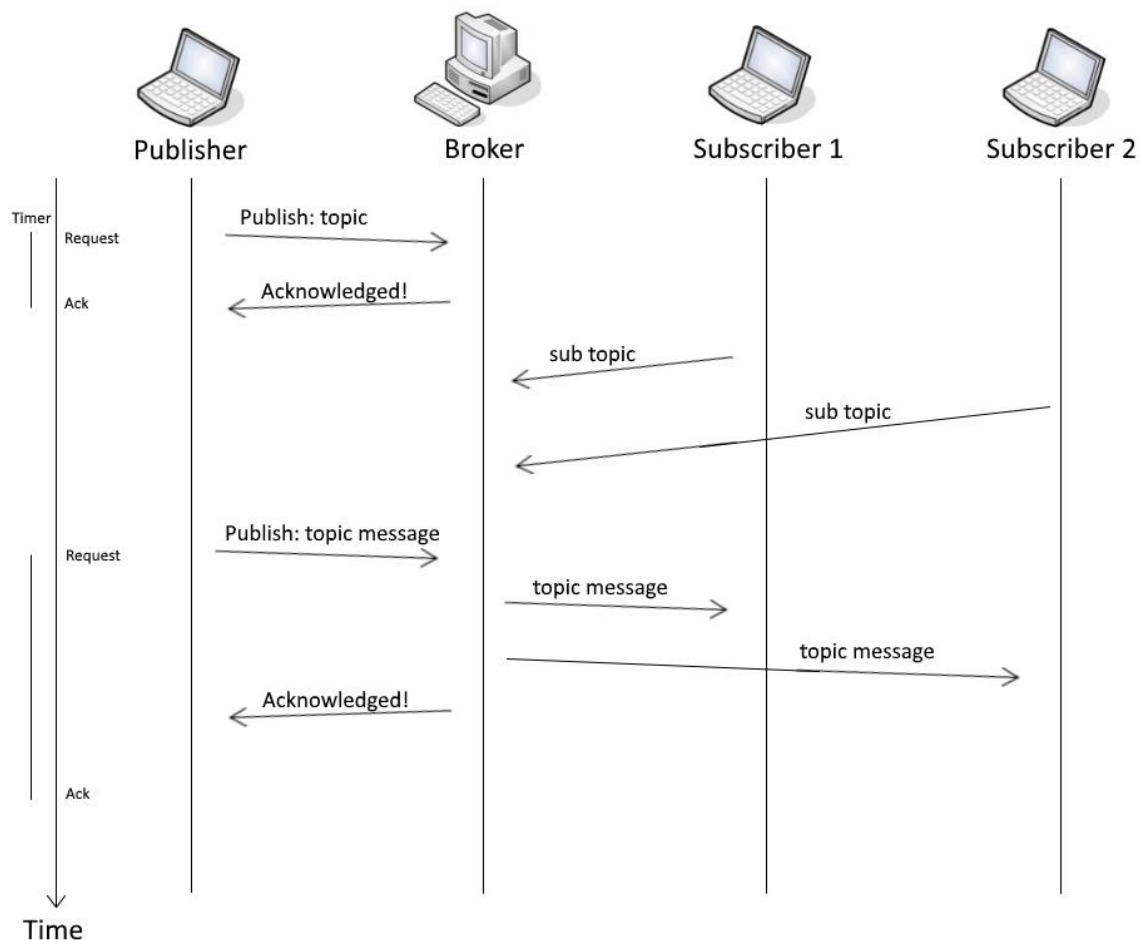


Figure 2: The flow of the packet exchange in my implementation. Publisher starts off by publishing a topic. The broker creates that topic and sends back an acknowledgment. Subscribers are now allowed to subscribe to this topic, they let the broker know if they want to do so. If a publisher publishes a message to the topic, and the subscriber list is not empty, the broker forwards the packet from the publisher to each subscriber in the list.

2.1 Design decisions

There were many design decisions to be made throughout the project which affected how the system worked.

- A subscriber cannot subscribe to a non-existent topic.

This is purely a design choice. As stated above in section 2, it is the publishers who decide which topics are created. This could easily be flipped around to allow the subscribers to create new topics when they attempt to subscribe to a non-existing topic, however, to me it doesn't make sense for a subscriber to subscribe to a non-existent topic and so I didn't implement my system this way.

- Subscribers only receive published messages from point of subscription.

To me it made more sense for this to be the case, as opposed to subscribers receiving a full history of messages even from a period before they were subscribed to a topic.

- Topics are limited to not having whitespace characters in them

This means that “validTopic” is valid, but “invalid topic” is not. The reason behind this is how the broker reads in packet contents. The broker turns the packet content into a string, and splits the string using the whitespace character ‘ ‘. Meaning that in the case of “invalid topic”, the broker will take the word “invalid” as the topic name, and the word “topic” as a message.

- Random port numbers for publishers and subscribers.

My implementation allows for multiple subscribers and publishers to be active at the same time through the use of different threads. Each subscriber and publisher must have a unique port number. I initially attempted to have these in increasing order, but I was unable to get that to work. As a result, I settled for the use of `Math.random()` to get a unique port number; because of the size of the range of the function, there is a close to zero probability of the same port number appearing more than once.

- Header of 2-byte overhead.

Originally, I wanted just 1-byte overhead for a code, to allow the broker to recognise different incoming packet types. However, based on how the broker analyses the packets I had to implement a whitespace character ‘ ‘ as one of the bytes of overhead. More details on this in section 3.1 Header design.

- No negative acknowledgments

The relationship between subscribers and broker doesn’t implement any error control in the case of any packets getting lost. The relationship between publishers and broker is Stop-and-Wait and implements a 1000ms timer, meaning there is no need for negative acknowledgments. However, there is no way for the broker to know what the next expected frame is.

3 Packet

The system is designed to exchange packets between publishers, broker and subscribers. To do this I used the `Java.net.DatagramPacket` class. As I mentioned in section 2.1, I had to use `Math.random()` for the socket addresses of the publishers and subscribers in order to make them unique.

My packets produce an overhead of 2 bytes per packet because of the header. Also, every time a packet is sent, it sends a fully sized packet of 65,536 bytes even if not all of those bytes are used – which isn't the most efficient solution.

3.1 Header design

A header was required for the packets in order to allow the broker to distinguish which packets from publishers and which are from subscribers.

The solution I came up with was to implement a code into the first byte of the header. Every time the broker receives a packet, it checks the first byte, and this allows it to know what to do next.

These codes used are:

- **1** = Message from publisher
- **2** = Subscribe new user to topic
- **3** = Unsubscribe user from topic
- **4** = Return a list of currently existing topics

As mentioned in section 2.1, the broker analyses packets by splitting the content based on the whitespace character ' '. This means that for the broker to distinguish the header code from the topic name, I had to implement a whitespace character as the second byte of the header. It is an extra byte of redundant overhead but in return it allows the system to work without having to redesign how the broker works.

4 Publisher

The publisher accepts a topic name and a message as input and sends on a packet with header code 0 to the broker. The implementation allows for multiple publishers to exist and for each publisher to publish to multiple topics. The publishers and the broker use a Stop-and-Wait approach to their communication.

4.1 Syntax

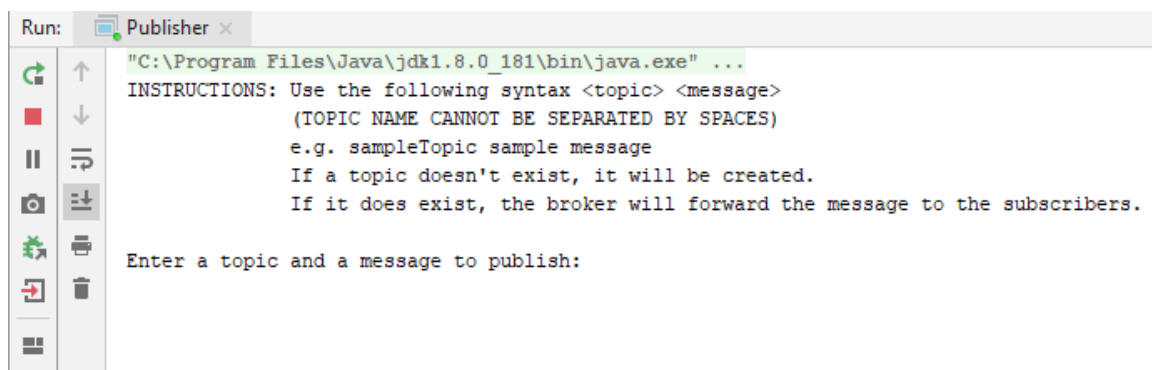


Figure 3: The instructions printed to the screen whenever a new publisher is launched.

```
Enter a topic and a message to publish: sampleTopic
Acknowledged!

Enter a topic and a message to publish: sampleTopic message
Acknowledged!
```

Figure 4: Sample input. First, the topic sampleTopic is created, then the message “message” is sent out to the subscribers of sampleTopic. The broker sends back the message “Acknowledged!” before the next input is possible.

4.2 Implementation

- The publisher can send messages along with a new topic however these messages will be discarded, as when the broker creates the new topic it will detect that there are no subscribers to the newly created topic.
- Stop-and-Wait relationship with the broker.
- Publishers require acknowledgment from broker before they can send another packet.
- Any publisher can publish to whatever topic it desires since the publishers don't keep track of any topics, it is the broker that handles all the traffic. It makes no difference to the broker which publisher the packet came from.
- Many publishers are allowed to exist in parallel as a result of multithreading. Every new publisher is given its own thread to run on without interrupting the others.

4.3 Stop-and-Wait

The publisher uses a stop-and-wait approach in the communication between with the broker. It sends out a packet containing the topic and a message, starts a timer and waits until it receives an acknowledgment from the broker. If the timer runs out before an acknowledgment is received, the publisher will re-transmit the packet and wait again. This process will loop until an acknowledgment is received, which allows it to take input again and send the next packet. The publisher does not let the broker know anything about the next expected packet, only the one it is sending.

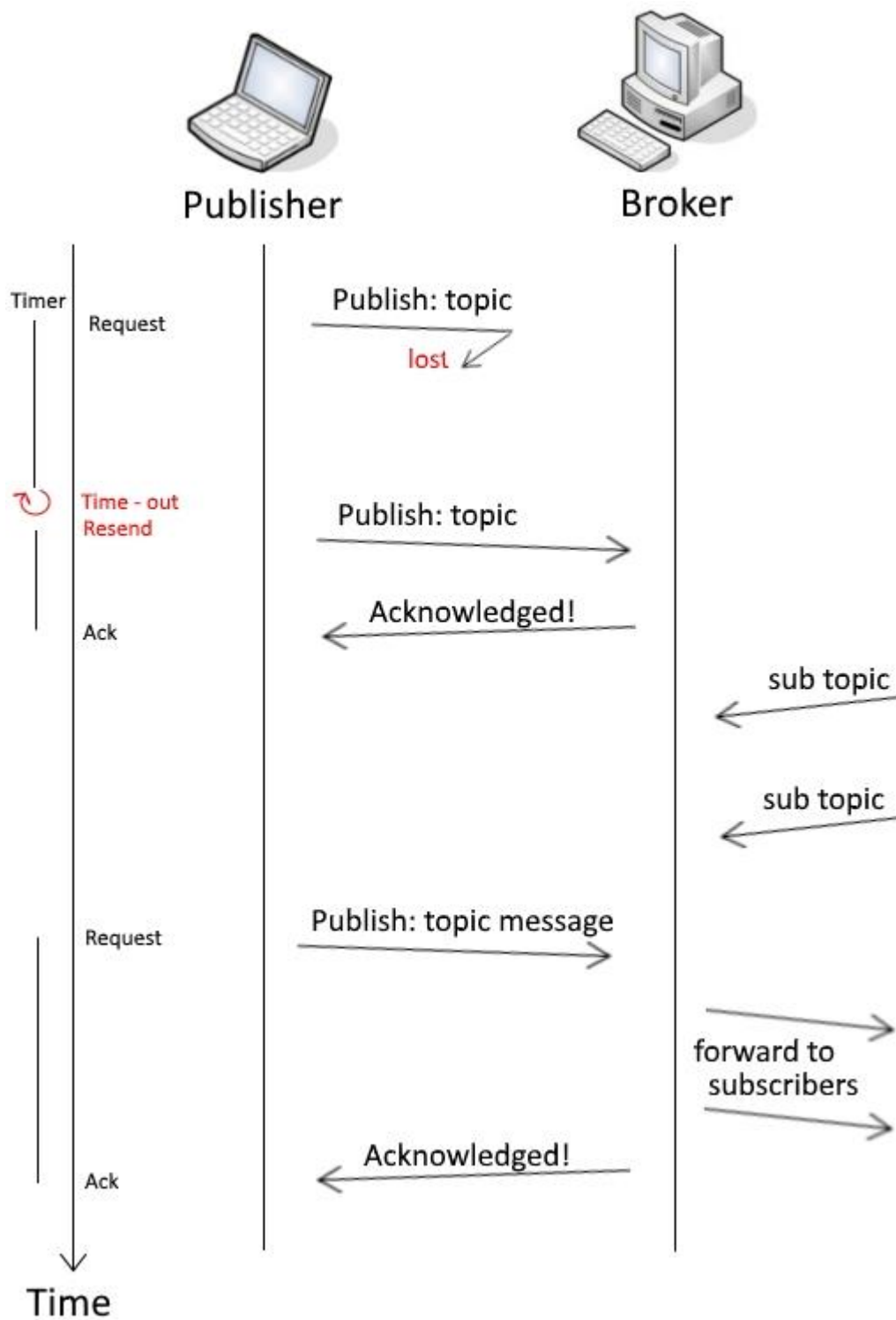


Figure 5: Stop-and-Wait relationship between publisher and broker. If the publisher doesn't receive an acknowledgment before the timer times out, then it will re-send the packet.

5 Broker

The broker is the centre piece of the system. It handles all the communication that happens and allows for a relationship between publishers and subscribers to exist.

In my implementation of the system, there is only one broker, as I didn't feel the need for there to be more than one as the project is on a small scale.

On a larger scale, more than one brokers could be used to split the data handling between multiple brokers and prevent overflow of any one broker. Multiple brokers also provide backup in case something goes wrong.

The broker does not take any direct input from the user. It outputs to the console sequentially what it is happening in the system.

5.1 Overall design

The main function of the broker is analysing packets and finding if anything needs to be done once a packet is received. In order to distinguish different types of packets, I used a code in the first byte of the packets. There is one code for a packet from a publisher, and the rest of the codes relate to subscriber packets.

To get the topic from a packet, the broker turns the packet content into a String, splits the String on the whitespace character ' ' and gets the second element of the array returned by the split function.

(The first element of this array is always the header code of the packet.)

This is why topics are limited to not having a whitespace character in them.

The broker contains a HashMap of type:

`< String, ArrayList < InetAddress > >`

It stores the topic name as a String key and creates an ArrayList of subscriber addresses for each topic. It then updates this ArrayList based on whether subscribers unsubscribe from the topic or if new subscribers subscribe to it.

5.2 Relationship with publishers

As mentioned previously, publishers and the broker implement Stop-and-Wait communication between each other. If the broker detects that the packet contains header code 0, i.e. is from a publisher, the broker checks if the topic contained in the packet is already a key in the HashMap.

- If the topic already exists, the broker checks if there are any subscribers in the ArrayList, if there are – forward the packet to their addresses and return an acknowledgment to the publisher.
- If the ArrayList is empty, there are no subscribers to an existing topic, meaning the broker doesn't have to forward the packet to anyone, and just returns an acknowledgment to the publisher.
- If the topic doesn't exist, i.e. is not in the brokers HashMap, the broker adds that topic to its map and creates an ArrayList for it. As it's a new topic, this means there are no subscribers to it yet and so the broker doesn't have to forward the packet, and so just returns an acknowledgment to the publisher.

5.3 Relationship with subscribers

The relationship between the broker and the subscribers is one dominated by the subscribers. They can send a request to a broker at any given time, the broker only responds to incoming subscriber requests, never initiates the contact. The subscriber can subscribe to a new topic, unsubscribe from a topic or request a list of existing topics from the broker. All of these commands have their unique header code to allow the broker to identify them and take corresponding actions. I will go into more detail about each command in the section 6 Subscriber.

5.4 Sample input and output

The figure consists of three screenshots of a Java application interface, showing the interaction between a Publisher, a Broker, and a Subscriber. Each screenshot has a 'Run:' tab at the top with three sub-tabs: 'Publisher', 'Broker', and 'Subscriber'. The interface includes a command line with various icons (run, stop, pause, etc.) and a text area for output.

Screenshot 1 (Top): The 'Publisher' tab is active. The output shows the following instructions and user input:

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...  
INSTRUCTIONS: Use the following syntax <topic> <message>  
               (TOPIC NAME CANNOT BE SEPARATED BY SPACES)  
               e.g. sampleTopic sample message  
               If a topic doesn't exist, it will be created.  
               If it does exist, the broker will forward the message to the subscribers.  
  
Enter a topic and a message to publish: topic1  
Acknowledged!  
  
Enter a topic and a message to publish: topic1 sample message  
Acknowledged!  
  
Enter a topic and a message to publish:
```

Screenshot 2 (Middle): The 'Broker' tab is active. The output shows the following messages:

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...  
Waiting for contact:  
  
Created new topic: topic1  
  
New subscriber to topic: topic1  
  
Message published to topic1 subscribers.
```

Screenshot 3 (Bottom): The 'Subscriber' tab is active. The output shows the following instructions, user input, and received message:

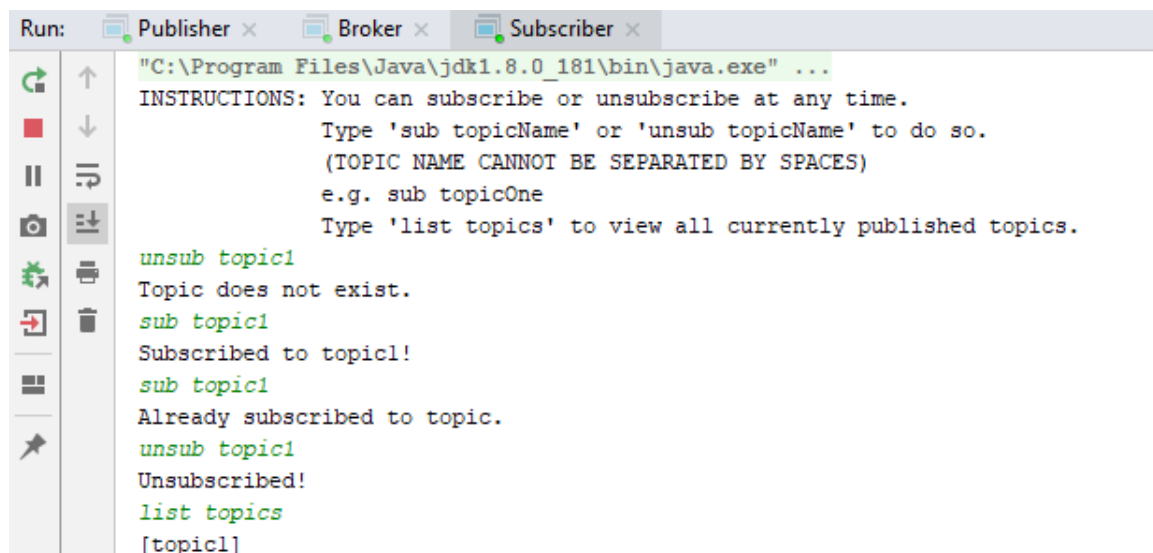
```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...  
INSTRUCTIONS: You can subscribe or unsubscribe at any time.  
               Type 'sub topicName' or 'unsub topicName' to do so.  
               (TOPIC NAME CANNOT BE SEPARATED BY SPACES)  
               e.g. sub topicOne  
               Type 'list topics' to view all currently published topics.  
  
sub topic1  
Subscribed to topic1!  
  
topic1: sample message
```

Figure 6: Publisher publishes topic, subscriber subscribes to it and receives published message. Broker prints what is happening in the system.

6 Subscriber

The subscribers in the system receive input from the user of what topic they should subscribe to or unsubscribe from. A subscriber can be subscribed to multiple topics at a time. The user may input a command to the subscriber at any given time.

6.1 Syntax



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
INSTRUCTIONS: You can subscribe or unsubscribe at any time.
                Type 'sub topicName' or 'unsub topicName' to do so.
                (TOPIC NAME CANNOT BE SEPARATED BY SPACES)
                e.g. sub topicOne
                Type 'list topics' to view all currently published topics.

unsub topic1
Topic does not exist.
sub topic1
Subscribed to topic1!
sub topic1
Already subscribed to topic.
unsub topic1
Unsubscribed!
list topics
[topic1]
```

Figure 7: The instructions given when a new subscriber is launched. Along with different types of input and output.

Valid commands for subscriber include:

- `sub topicName`
Subscribes the subscriber to topic `topicName`, if it exists.
- `unsub topicName`
Removes user from subscriber list of `topicName` if `topicName` is an existing topic.
- `list topics`
Prints currently existing topics as String.

6.2 Implementation

The Subscriber class uses Java's implementation of multithreading to allow multiple subscribers to exist in parallel. To allow them to co-exist, they must have unique port numbers, which are assigned using the `Math.random()` function.

A subscriber may subscribe to many topics at once and has the possibility of unsubscribing from them as well. It can also request a list of all currently existing topics from the broker. Each of these commands has a unique implementation which I shall discuss now.

6.3 Subscribe

A subscriber can request to subscribe to a topic through the input:

`sub topicName`

This command puts header code 2 into the packet along with the topic name as data. This packet is sent on to the broker. The broker detects the code in the first byte and knows that this is a subscription request. It then checks if it contains the topic in the packet in its `HashMap`.

- If it does, the subscribers `InetSocketAddress` is added to the corresponding `ArrayList`. If the subscription is successful, the broker sends back a packet to the subscriber with the message "Subscribed to topicName!".
- If the topic requested does not exist, the broker sends back a packet to the subscriber informing them that "Topic does not exist."
- If a subscriber tries to subscribe to a topic it is already subscribed to, the broker sends back a message "You are already subscribed to this topic." and ignores the subscription request as it does not need to do anything.

6.4 Unsubscribe

A subscriber can request to unsubscribe from a topic through the input:

unsub topicName

This command puts header code 3 into the packet along with the topic name as data. This packet is sent on to the broker. The broker detects the code in the first byte and knows that this is a request to unsubscribe. It then checks if it contains the topic in the packet in its HashMap.

- If it does, the subscribers InetAddress is removed from the corresponding ArrayList. If the broker successfully unsubscribes the user, the broker sends back a packet to the user with the message “Unsubscribed!”.
- If the topic requested does not exist, the broker sends back a packet to the subscriber informing them that “Topic does not exist.”.
- If a user tries to unsubscribe from a topic which it is not subscribed to, the broker sends back a message saying, “You are not currently subscribed to this topic.”.

6.5 List topics

A subscriber can request a list of currently existing topics from the broker through the input:

topics list

This command puts header code 4 into the packet. It also puts “list” as a topic as a way to prevent the broker from having a null pointer exception when it is getting topic from received packet. This shouldn’t affect efficiency much and there are other ways to get around this problem, however this was the one I chose to go with.

The broker recognises the code in the header and creates a packet with the existing topics (keys in the HashMap) as a String and sends this packet back to the subscriber.

7 Reflection

There are many things which went right for me in this project, however there are still areas which could be improved. These include:

- Implement support for multiple brokers
- Implement sequence numbers in packets and create a queue system
- Improve the error control
- Not use `Math.random()` for port numbers (if possible?)

Overall, I'm really satisfied with this project and how it went. I learnt a lot while doing it and it deepened my understanding of communication systems.

Average time spent (including background research etc.) = ~ 25 hours

That's all folks!