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

CS2031 Telecommunication II
Assignment #1: Publish - Subscribe

Jerzy Jaskuc, Std# 17341645

December 28, 2018

Contents

1 Introduction **1**

2 Theory of Topic **1**
2.1 Packets 2
2.2 Flow control 2
2.3 Nodes 2

3 Implemenation **3**
3.1 Implementation of the Packets 3
3.2 Implementation of the Flow Control 4
3.3 Node Structure 4
3.4 User Interaction 5

4 Summary **5**

5 Reflection **6**

1 Introduction

The general task was to implement a system, where a Publisher would publish to a Broker and a Broker would forward these messages to a number of Subscribers, that subscribed to that topic. In order for this to work, it was needed to implement a protocol, which works similarly to the MQTT protocol.

2 Theory of Topic

Imagine a situation where a user want to subscribe to a certain topic of interest and recieve updates on it. Of course, user can subscribe directly to publisher, but what if we want to recieve updates not only from publishers we know, but from any publisher that publishes the topic of interest? What if publisher publish on a variety of topics and we are only interested in one? This is where broker is introduced to the system. When subscriber connects to a broker, he can subscribe to a variety of topics and the broker will store them. Then, when a connected publisher publishes something on a given topic and sends it to the broker, the broker forwards it to the subscribers that were interested in a given topic.

2.1 Packets

The most straightforward way to introduce such a system is to use MQTT like protocol, where exchange of packets is as shown below (see figure 1).

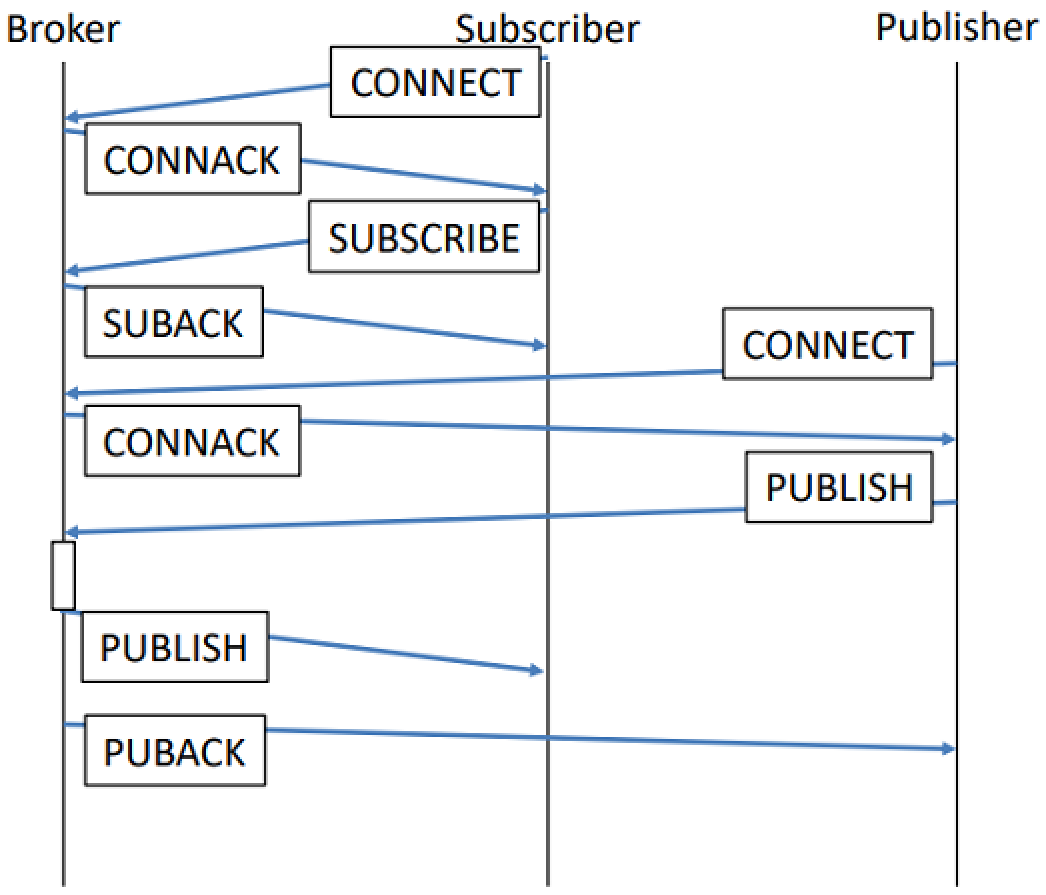


Figure 1: This diagram represents the packet exchange in typical MQTT implementation, and my system follow it.

In this case the system will require only *6 packet types*, which means it won't introduce too much overhead.

2.2 Flow control

In order to not lose the packets, the Flow Control is needed. The most two common ways would be:

- Stop and Wait ARQ
- Go-back-N ARQ, or it's improved version - Selective Repeat

Since *Go-back-N* is relatively hard to implement, as well as since we will be dealing with single packet transmissions at a time, I decided to stick with *Stop and Wait* ARQ. The way *Stop & Wait* work is quite simple, see figure 2.

2.3 Nodes

Each node in this implementation represent one of the 3 machines: *Subscriber*, *Broker*, or *Publisher*. *Publisher* publishes messages with certain topics for the *Subscribers* and sends it to the *Broker*. *Brokers* task is to forward these messages to the *Subscribers*. That's why it also maintains the list of topics and *Subscribers* that subscribed to any of them. *Subscribers* can subscriber to topics and then recieve a message from *Publishers*, forwarded by *Broker*, with the topics they subscribed to.

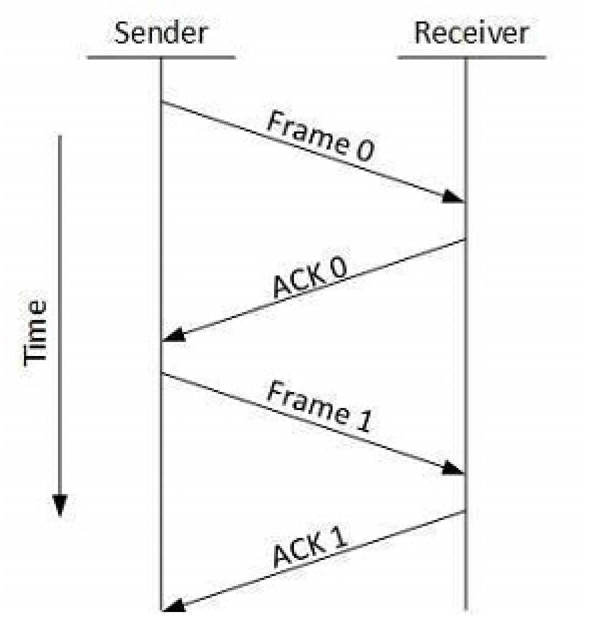


Figure 2: This diagram represents how Stop & Wait ARQ work. For every sent packet, acknowledgement is sent indicating that the packet was successfully recieved. Otherwise the packet will be resent after a certain time.

All of the machines will be on the localhost for easier implementation, and each will have it's unique port number on which it recieves and from which it sends the packets.

3 Implemenation

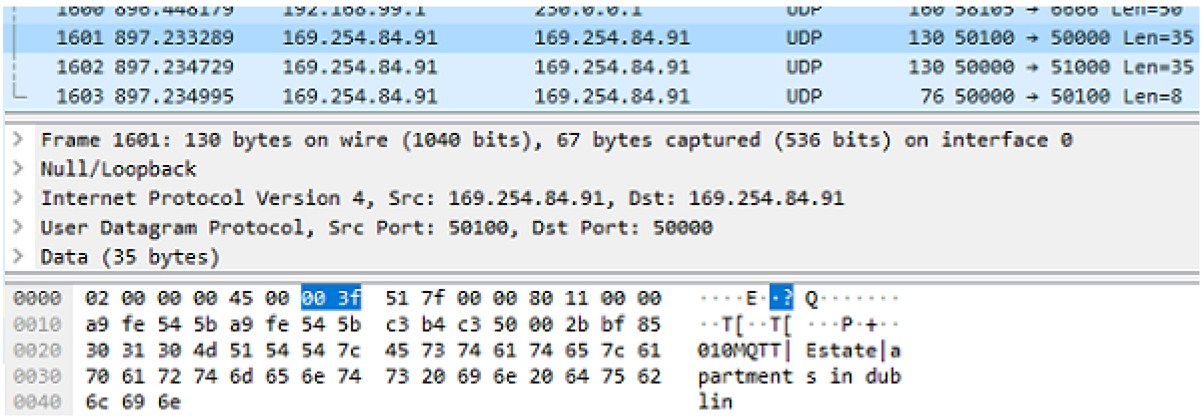
Generally, in my implementation, I decided to make it fully in Java and on a local host with different port addresses. This is to avoid general troubles that might arise when setting system up on a containers or virtual machines, as well as it gave me the ability to easily continue work on different machines without any problems. All of the port addresses are preconfigured, but can be easily changed if needed.

3.1 Implementation of the Packets

All of my data that is being exchanged is of type `String`, since it can be easily converted into `DatagramPacket` using `PacketConverion` class. In my implementation i use special character 'l' (ASCII value 124) as a delimiter. This allows relatively straightforwart separation of data, as I dont need to worry about length of the fields. However, this character becomes reserved and the users cannot use it in their messages, or it won't be processed properly. To define a packet type i use a `String` header with 3 *'bits'*, which is always before the first delimiter. The rest of the fields depend on the type of the packet. The packet types are as follows:

- `CONNECT` header - **000**
- `CONNACK` header - **001**
- `PUBLISH` header - **010**
- `PUBACK` header - **011**
- `SUBSCRIBE` header - **100**
- `SUBACK` header - **101**

The additional fields are only in `PUBLISH` and `SUBSCRIBE` packets. In the first case, the next field is the *topic* and the *message* after it. In the `SUBSCRIBE` case, the only additional field is the *topic*.
Here is an example of the `PUBLISH` packet, see figure 3.



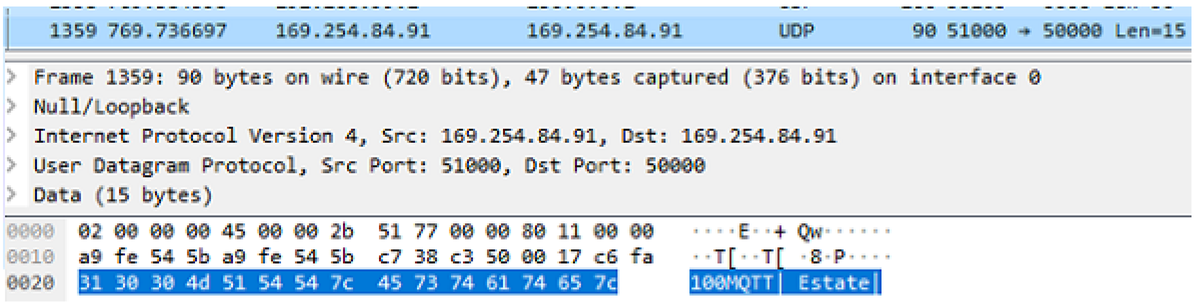
```
1601 897.233289   169.254.84.91   169.254.84.91   UDP   130 50100 → 50000 Len=35
1602 897.234729   169.254.84.91   169.254.84.91   UDP   130 50000 → 51000 Len=35
1603 897.234995   169.254.84.91   169.254.84.91   UDP    76 50000 → 50100 Len=8

> Frame 1601: 130 bytes on wire (1040 bits), 67 bytes captured (536 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 169.254.84.91, Dst: 169.254.84.91
> User Datagram Protocol, Src Port: 50100, Dst Port: 50000
> Data (35 bytes)

0000  02 00 00 00 45 00 00 3f  51 7f 00 00 80 11 00 00   ····E··? Q·······
0010  a9 fe 54 5b a9 fe 54 5b  c3 b4 c3 50 00 2b bf 85   ··T[··T[ ···P·+··
0020  30 31 30 4d 51 54 54 7c  45 73 74 61 74 65 7c 61   010MQTT| Estate|a
0030  70 61 72 74 6d 65 6e 74  73 20 69 6e 20 64 75 62   partment s in dub
0040  6c 69 6e                                           lin
```

Figure 3: This is an example of an exchanged PUBLISH packet registered by Wireshark.

And an example of the `SUBSCRIBE` packet, see figure 4.



```
1359 769.736697   169.254.84.91   169.254.84.91   UDP   90 51000 → 50000 Len=15

> Frame 1359: 90 bytes on wire (720 bits), 47 bytes captured (376 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 169.254.84.91, Dst: 169.254.84.91
> User Datagram Protocol, Src Port: 51000, Dst Port: 50000
> Data (15 bytes)

0000  02 00 00 00 45 00 00 2b  51 77 00 00 80 11 00 00   ····E··+ Qw······
0010  a9 fe 54 5b a9 fe 54 5b  c7 38 c3 50 00 17 c6 fa   ··T[··T[ ·8·P····
0020  31 30 30 4d 51 54 54 7c  45 73 74 61 74 65 7c      100MQTT| Estate|
```

Figure 4: This is an example of an exchanged SUBSCRIBE packet registered by Wireshark.

All of the packets have a limit of size which is *65536 bytes* . If the limit will be exceeded, the packet won't be processed correctly.

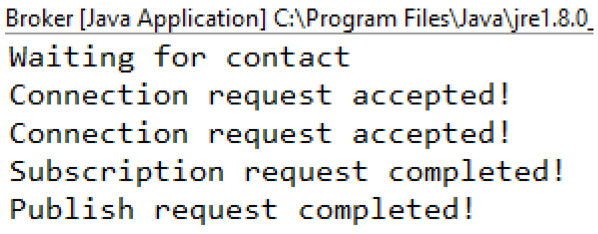
3.2 Implementation of the Flow Control

Since the message is limited to a packet size, there wasn't any need for more complicated flow control, therefore I just use Stop & Wait ARQ flow control method. The nodes use `Timer` class and `TimeoutTimer` class, which is my own extension of `TimerTask` class. Every time a packet is sent the current `Thread` gets stopped and not continued until the acknowledgement is recieved. If it is not recieved in *7 seconds*, the set `TimeoutTimer` will resend the packet. On the bright side of this implementation, no packets will be lost during exchange, and there's no additional overhead in the packet introduced. On the negative side, if connection is lost, the `Thread` won't be notified and the node will be resending, until the reciever get's back online or the program is manually terminated.

3.3 Node Structure

All of the nodes are build on `Machine` class, which basically contains a listener that recieves packets. While *Publisher* and *Subscriber* contain input console, the *Broker* class only outputs messages when certain operations occur. As previously mentioned, all port addresses are preconfigured, but not necessarily hardcoded. If the need arises the Sockets, which are constants, can be changed to any real and accesible value and if

the Socket is not taken it will be normally used. The the port limits in the constructor would need to be changed as well however.
When a packet is recieved, it is checked, which *Header* it contains. *Subscriber* and *Publisher* are sending `CONNECT` request to *Broker* upon start. If then acknowledgement is recieved, the connection is established.
For every `SUBSCRIBE` packet recieved, *Broker* stores the topics, to which *Subscribers* subscribed in a `HashMap`, where key is the Topic and the value is the `ArrayList` of `InetScoketAddresses` of *Subsrcibers* subscribed to a given topic. This works out perfectly, since topics by nature are unique, and multiple *Subscribers* can subscribe to a same topic.
When a `PUBLISH` is recieved on a *Broker* and if there's someone subscribed to recieved topic, it will forward the message with the topic to a *SUBSCRIBER*. It is worth mentioning that this is the only case where *Broker* does not wait for the acknowledgements. This is due to the fact that if there's a lot of *Subscribers*, if would require too much time to wait for acknowledgement from each of them, as well as it's quite hard to properly set up multiple threads running the same method at the same time.

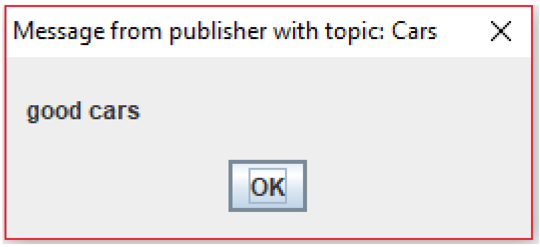


```
Broker [Java Application] C:\Program Files\Java\jre1.8.0_
Waiting for contact
Connection request accepted!
Connection request accepted!
Subscription request completed!
Publish request completed!
```

Figure 5: This is an example of the Broker console output after exchange of some of the packets.

3.4 User Interaction

To run the system, each node have to be *manually* started. However, as many nodes as needed, of each class, can be started.
For *Publisher* and *Subscriber* classes, the console also takes input from the user. However, in *Publisher* the console is inside of the class and within the main `Thread`. In *Subscriber* the input is in a separate class and `Thread`. That's because *Subscriber*, unlike *Publisher*, also recieves messages and the use of the default console with the `Scanner` class will stop the `Thread` until it will recieve the input. That prevented the message to be printed at the time of the reciept. In addition, when recieving the message, the user might be writing something. In order to prevent interrupting him, I use `JOptionPane` to display the message. See figure 6.



Message from publisher with topic: Cars

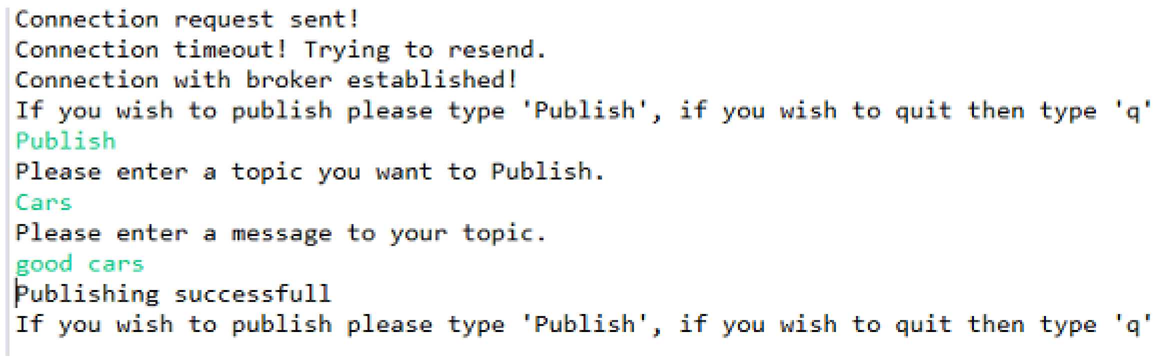
good cars

Figure 6: Example of the recieved message on a Subscriber.

To every message, or topic too in case of *Publisher*, the special character 'l' is automatically added as a delimiter for later processing.
Here are some examples of the consoles on the *Publisher* (see figure 7) and *Subscriber* (see figure 8).

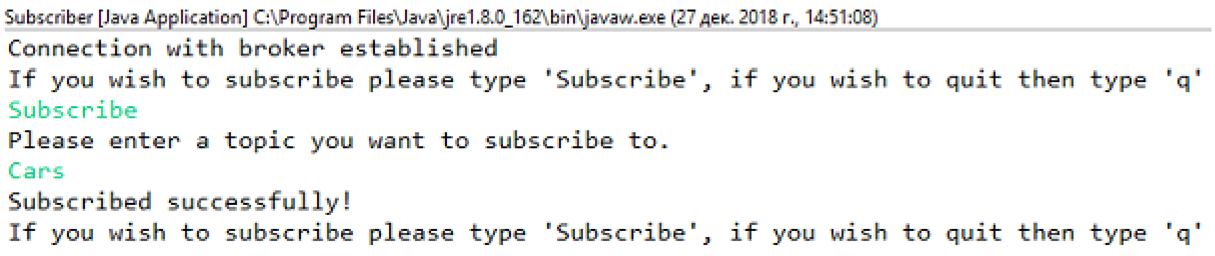
4 Summary

Summarizing, this implementation of **Publish - Subsrcibe** uses modified MQTT protocol and packet types to work. For the flow control part Stop & Wait ARQ was used on most of the packet exchange. The



```
Connection request sent!
Connection timeout! Trying to resend.
Connection with broker established!
If you wish to publish please type 'Publish', if you wish to quit then type 'q'
Publish
Please enter a topic you want to Publish.
Cars
Please enter a message to your topic.
good cars
Publishing successfull
If you wish to publish please type 'Publish', if you wish to quit then type 'q'
```

Figure 7: An example of a Publisher console after exchange of some of the packets



```
Subscriber [Java Application] C:\Program Files\Java\jre1.8.0_162\bin\javaw.exe (27 дек. 2018 г., 14:51:08)
Connection with broker established
If you wish to subscribe please type 'Subscribe', if you wish to quit then type 'q'
Subscribe
Please enter a topic you want to subscribe to.
Cars
Subscribed successfully!
If you wish to subscribe please type 'Subscribe', if you wish to quit then type 'q'
```

Figure 8: And example of a Subscriber console after exchange of some of the packets

implemetation provides nodes which support multithreading, are not over-hardcoded and provide relatively straightforward and rather intuitive user experience. The system is also quite stable and not prone to crashing.

5 Reflection

As the first big assignment like that I feel that I did a good work with. I've spent around 30-40H total on it and managed to implement most of the major features i wanted it to have. For the workload I divided it, so overall it took me around 2 months to finish this assignment. I feel like I structured it well, however I think i should do more designing without coding parts in the next assignment, since i got lost in what i wanted to do at a times. There are still ways to improve this system, such as setting limits to the input, flow control when forwarding `PUBLISH` packets, make an *'unsubscribe'* option ect. But i feel like this system is pretty solid with what it is and it's time to move to the next assignment.
This assingment will also go on github.com/EZCodes after the assignment deadline.