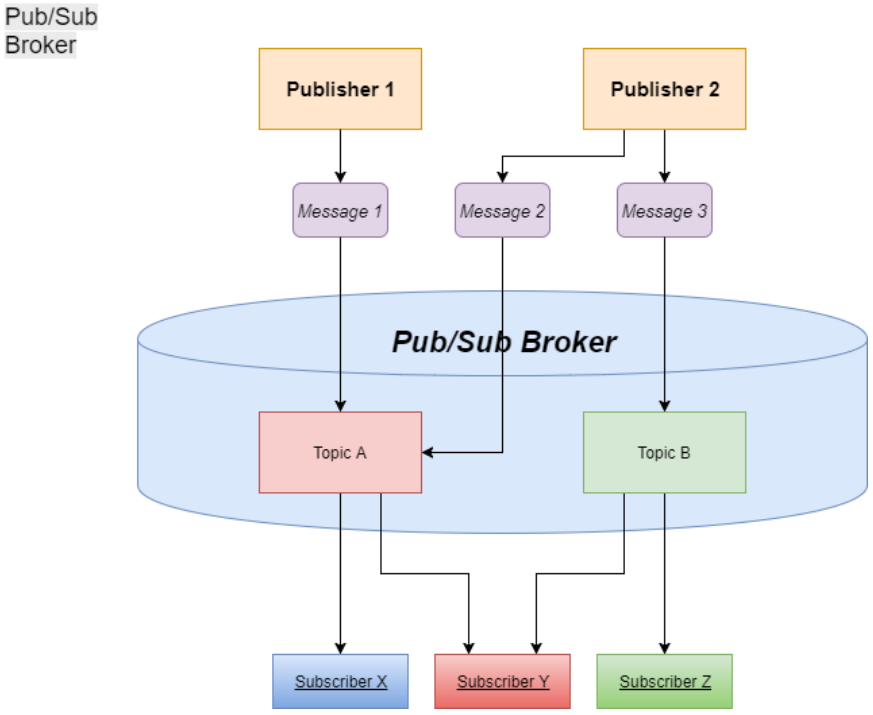
Programming Assignment

In this project we will be creating a simple version of a Publish-Subscribe (Pub/Sub) system, similar to Apache Kafka or RabbitMQ. For more general information on Pub/Sub systems you can read this article from Google and/or watch this lecture that describes the Pub/Sub systems, their functionality, and the differences between different implementations.

In this project, we will be implementing a simple version of those systems. More specifically, we will implement three components: a Publisher that can run with multiple instances, a Subscriber that can also run with multiple instances and a Pub/Sub Broker that will be handling the message relaying between Publishers and Subscribers. Our architecture will be similar to that of the following picture:



You can use any programming language of your choice to implement this project.

However, you cannot use existing projects or libraries implementing a Pub/Sub system that are available on the Web.

It is okay to get some helper function or some small functionality from somewhere but, if you do so, please cite your sources.

Again, it is not okay to clone a git project or download it from somewhere else implementing a Pub/Sub system and submit it as your project. If you do, that constitutes plagiarism and you will fail the class.

1. **Subscriber (20%)**

The Subscriber is a program that, when started, will connect to the Broker and subscribe to one or more topics. The Subscriber should run as follows:

```
subscriber -i ID -r sub_port -h broker_IP -p port [-f command_file]
```

where
`-i` indicates the id of this subscriber. This id will be sent to the broker so that it can keep track of the subscriber and print messages accordingly.
`-r` indicates the port of this specific subscriber. The IP address of the subscriber is implied to be that of the current machine the subscriber is running on.
`-h` indicates the IP address of the broker.
`-p` indicates the port of the broker
`-f` is an optional parameter that indicates a file name where there are commands that the subscriber will execute once started and connected to the broker, before giving control to the user from the keyboard.

Here is an example command file `subscriber1.cmd`

```
2 sub #hello
10 sub #world
300 unsub #hello
25 sub #great
```

The file has 3 columns. The first column represents the number of seconds that the subscriber should wait after connecting in order to execute that command. This number should be greater or equal to 0. The second column represents the command to execute: `sub` for subscribe and `unsub` for unsubscribe. The third and final column represents the topic that the subscriber is interested in subscribing in, or, unsubscribing from. Note that the commands are sequential, i.e. the line `10 sub #world` will get executed after a total of 12 seconds (due to the first line needing 2 seconds) after connecting to the broker.

Here is an example of how we could run the subscriber:

```
subscriber -i s1 -r 8000 -h 127.0.0.1 -p 9000 -f subscriber1.cmd
```

In this case, we start a subscriber process with the id s1. It will connect to the broker locally, i.e. at the 127.0.0.1 IP address, and at port 9000. The file we saw above will get executed. This means that, after connecting, the subscriber will wait for 2 seconds, then subscribe to the topic #hello, then it will wait for 10 seconds, will subscribe to #world, then after 5 minutes will unsubscribe from #hello and, finally, after 25 seconds will subscribe to #great. Note that after unsubscribing from a topic the subscriber should no longer receive messages on that topic from the broker.

After the command file is processed, the subscriber waits for more commands from the keyboard. The commands are of the same format as in the command file.

Once a command is received (either from a file or from the keyboard) the subscriber sends the respective command to the broker over the connected socket, with some additional info. The commands sent to the broker from the subscriber are of the form:

```
SUB_ID COMMAND TOPIC
```

In our case, for the example file that we saw above the subscriber will send the following commands (after the necessary waits) to the broker:

```
s1 sub #hello
s1 sub #world
```

```
s1 unsub #hello
s1 sub #great
```

These commands are sent one at a time and the subscriber waits to receive an OK message before sending the next command.

The broker (described below) needs to record what topics the subscriber has subscribed in and, when there is something published in this topic, it needs to forward that message to the subscriber. For example, if there is a message `today is a nice day` published in the topic `#hello` then this message will be forwarded to the subscriber by the broker and the subscriber should print it out in the console as follows:

```
Received msg for topic #hello: today is a nice day
```

You should be able to start multiple subscribers with different IDs (and ports), and all should connect to the Broker, subscribe accordingly and see the results.

2. **Publisher (20%)**

   The Publisher connects to the Broker in a similar way to the Subscriber. Except it is not consuming messages like the Subscriber but, instead, it is producing (i.e., publishing) these messages. The Publisher should run as follows:

```
publisher -i ID -r sub_port -h broker_IP -p port [-f command_file]
```

where
`-i` indicates the id of this publisher. This id will be sent to the broker so that it can keep track of the publisher and print messages accordingly.
`-r` indicates the port of this specific publisher. The IP address of the publisher is implied to be that of the current machine the publisher is running on.
`-h` indicates the IP address of the broker.
`-p` indicates the port of the broker
`-f` is an optional parameter that indicates a file name where there are commands that the publisher will execute once started and connected to the broker, before giving control to the user from the keyboard.

Here is an example command file `publisher1.cmd`

```
3 pub #hello This is the first message
5 pub #world This is another message
10 pub #hello One more message
```

The first column in the file represents the number of seconds that the publisher should wait after connecting in order to execute that command. This number should be greater or equal to 0. The second column represents the command to execute: this will always be `pub` for the publisher. The third column represents the topic that the publisher will publish to.

All topics both for publisher and subscriber are one keyword. They don't need to start with # it's just easier from a user point of view.

The remainder of the line after the topic (i.e. the first keyword after `pub`) is the message that the publisher will send to the broker.

Here is an example of how we could run the publisher:

```
publisher -i p1 -r 8200 -h 127.0.0.1 -p 9000 -f publisher1.cmd
```

In this case, we start a publisher process with the id p1. It will connect to the broker locally, i.e. at the 127.0.0.1 IP address, and at port 9000. The file we saw above will get executed. This means that, after connecting, the publisher will wait for 3 seconds, then publish to the topic #hello the phrase This is the first message, then it will wait for 5 seconds, will publish to #world, then after 10 seconds will publish again to #hello.

After these commands are executed, the publisher also waits for more commands from the keyboard. The commands are of the same format as in the command file.

Once a command is received (either from a file or from the keyboard) the publisher sends the respective command to the broker over the connected socket, with some additional info. The commands sent to the broker from the publisher are of the form:

```
SUB_ID COMMAND TOPIC MESSAGE
```

In our case, for the example file that we saw above the subscriber will send the following commands (after the necessary waits) to the broker:

```
p1 pub #hello This is the first message
p1 pub #world This is another message
p1 pub #hello One more message
```

These commands are sent one at a time and the publisher waits to receive an OK message before sending the next command.

The broker (described below) needs to identify all the subscribers that have subscribed to the specific topic and forward the message to all of them. record what topics the subscriber has subscribed in and, when there is something published in this topic, it needs to forward that message to the subscriber. For example, if there is a message `today is a nice day` published in the topic `#hello` then this message will be forwarded to the subscriber by the broker. Once the publisher receives an OK message from the broker it should print it out in the console as follows:

```
Published msg for topic #hello: today is a nice day
```

You should be able to start multiple publishers with different IDs (and ports), and all should connect to the Broker, and publish accordingly.

3. **Broker (50%)**

The Broker connects the Publishers and the Subscribers together. Its job is to remember all the topics that each subscriber has subscribed to and, when a publisher publishes something about a specific topic, it should forward the message to all subscribers that have subscribed to that topic.

As an example, if there are 3 subscribers that have subscribed to topic #hello, then once a publisher publishes something for that topic, all three subscribers need to receive the message and they should all print it out (as we already described above).

The Broker should run as follows:

```
broker -s s_port -p p_port
```

where
`-s` indicates the port of this specific broker where subscribers will connect. The IP address of the broker is implied to be that of the current machine the publisher is running on.
`-p` indicates the port of this specific broker where publishers will connect. Again, the IP address of the broker is implied to be that of the current machine the publisher is running on.

Once the broker starts, it expects connections from Publishers and Subscribers. Once either of them is connected, it receives the commands that we described in the respective sections for Publishers and Subscribers above and processes them.

For example, if a subscriber has sent a command such as `s1 sub #hello` then the broker needs to remember this information, so when a publisher sends a command such as `p1 pub #hello This is the first message` it can forward the message accordingly. Note that the broker needs to send the message to **all** subscribers that have subscribed to the topic. Once a command is processed either from a subscriber or a publisher, the broker sends back an OK string. If a subscriber unsubscribes from a topic then it should not receive relevant messages any more.

Note also, that the broker does not store messages. In other words if a subscriber subscribes to a topic after some messages have been published to that topic then the subscriber will miss them. The Broker relays messages only to subscribers that are subscribed when the message is published. Remembering messages and eventual consistency is out of the scope of this project.

4. **Multiple Publishers/Subscribers (10%)**

The broker should be able to handle multiple publishers and subscribers. For consistency among projects assume that up to 5 publishers and up to 5 subscribers can be connected at one time.

Deliverables

- All files for your project need to be zipped in **one single zip file.** The file name should be in the following format: Lastname_Firstname_Project.zip
- Please upload the zip file to eClass (look for announcement) by the deadline. **There are absolutely no exceptions and no extensions.**
- Inside the zip file, please include a) your source code b) any results that are part of the project c) a README text file that explains how to compile (if needed) and run your code and d) a short and concise report (preferably pdf - no more than 10 pages) describing your work and your approach to solving the project. **Please do not include binaries, nor the original or any derived data in the zip file.**
- **Please ensure that your instructions for compiling and running your code are clear.**
- **Your code will be compiled from scratch.**

Important Notes

- Please check the eClass website https://eclass.uoa.gr/courses/DI508/ : regularly for announcements. Announcements will show up there and will also be sent to the mailing list. Also, please sign up to piazza.com/uoa.gr/spring2022/m111 where there can be discussion and potential clarifications regarding the project.
- You are free to make any assumptions you may need to along the way as long as you document them in your report.
- The project is meant to be worked on by the student submitting and only that student. In the event that there is a submission not worked on by the student, then the student fails the class.
- Copying the project from existing sources (e.g., the Web, github, etc.) is strictly not allowed. If you use small pieces of code (a few lines) from somewhere please provide a reference to the source.
- You are responsible to safeguard your code, so if you are using a code repository service (e.g., github) keep your repository private. Cases when code was cloned and was also submitted by someone else cannot be reliably traced and result in everyone involved (regardless of who is at fault or who copied from whom) failing the class.
- Although students are expected (and encouraged) to chat among them on potential solutions and approaches, sharing code and solutions is strictly not allowed. In the event that two or more students provide submissions that have common pieces, everyone involved (regardless of who is at fault or who copied from whom) fails the class.