Faculté des Sciences appliquées

INFO0010-4 : Introduction to Computer Networking

Socket Programming : Broker Implementation

Teacher :
Guy LEDUC

Assistant :
Emeline MARECHAL

Group :
Romain LAMBERMONT
Arthur LOUIS

December 17, 2021

Contents

1 Introduction

In this project, we implemented a Broker to handle MQTT messages to communicate with IoTs sensors and Subsribers to implement the Monster Hunting Game. In this small report, we'll explain the architecture of our implementation, what class we have used and for what purpose. We'll then reflect on MQTT quality of service. We'll finally check what are the limits of our implementation and what we could have done better, to implement new features in our MQTT management.

2 Software architecture

When starting our project, we had no idea of how to implement the Broker, so we began to sketch our architecture, representing the Broker, IoTs sensors and Subscriber. We then started to add our classes one by one to make everything communicate. The sketch made it easier to implement everything we wanted and what we should do in each class to make the Game run on the Subscribers. The architecture is as follows, to start we launch a **Broker**, when a new connection enters, it creates a new **Client** and we handle it with **Messages** that we can **Receive** and **Send**. To remember what a client is interested about, there are **Topics** a client can subscribe to.

Here is the final sketch of our implementation :

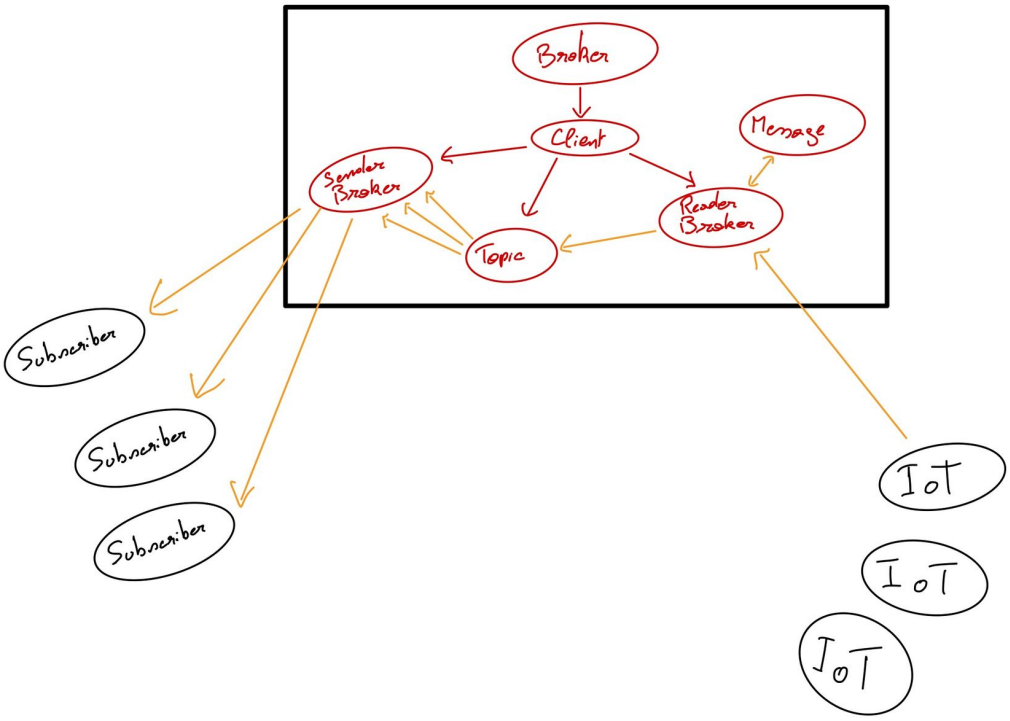

Figure 1: Publish message handeld by our implementation

To conclude this first section, here is a reminder of our classes :

- **Broker.java** : Accepts new connections using a ServerSocket and instantiates each needed class. After accepting a connection, it delegates the corresponding Client to the **Client** class.
- **Client.java** : Receives a Socket and divides in two to **Send** and **Receive**.
- **ReaderBroker.java** : Receives packets and handle them to create the corresponding messages to **Send**.
- **SenderBroker.java** : Sends the response of received packets handeld by **ReaderBroker.java**.
- **Topic.java** : Stores the **Clients** interested in specified topics.
- **Message.java** : MQTT toolbox used to decode **received** packets and to compose new ones to **send**.

3 Reflection on MQTT's quality of service

MQTT uses TCP/IP as the protocol to exchange packets and in TCP, there's no loss or alteration of the transmitted data. However, if the TCP connection closes (e.g crash or client restart) a totally new TCP connection needs to be created. Without QoS 1 (at least once delivery, using PUBACK), in case of a crash of a connection, the conversation between server and client would need to restart from zero but in the case of a MQTT QoS 1 (and Persistant Session), the conversation can keep on as if nothing happend. But we have to be careful using QoS 1 because the implemantation needs to be able to handle duplicates (those can be harmful for the Subscribers). With QoS 2, we again improve the implemantation by guaranteeing the transmit of a packet with no duplicates (by using PUBREC, PUBREL, PUBCOMP). This QoS 2 feature is great for an implementation that can keep the conversation on in case of crash while being careful of duplicates.

4 Limits & Possible Improvements

In this implementation, we created the simplest MQTT Broker possible. Indeed, there's no advanced features that are available in MQTT protocol, namely, Persistant Session, Retained Messages and the Last Will and Testament (our Broker is only QoS 0). However, this Broker is only used locally (localhost), so the risk of loss connection is almost zero. The more advanced features mentionned above would be necessary if our Broker was used online or with a different IoTs/Subscribers structure.