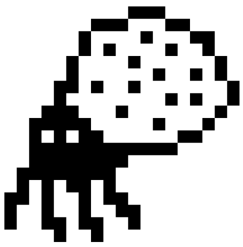
INFO0010 Introduction to Computer Networking
Second part of the assignment

E. Marechal, G. Leduc
Université de Liège

Academic year 2021-2022

Submission before 17th of December 2021



Abstract

In this assignment, you will have to implement a client-server application using Java Sockets.
The application is the same as the one for the first part of the assignment, namely, the game of Monster Hunting. This time however, you will have to implement the server (instead of the client), and the communication between the various parties will be ensured with the MQTT Protocol, the standard for IoT messaging.
Students will work in teams of 2 for this part of the assignment.

Contents

1 Monster Hunting with Internet of Things (IoT)

1.1 Reminder

This section is a small reminder of the Monster Hunting application that was presented in the first part of the assignment. The game relies on sensors distributed across the environment (a simple NxN Grid), and that are able to detect when a monster is close to them. The sensors share the data they collect by connecting to the main server, which uses the publish/subscribe message pattern. A client can then connect to the server and subscribe to receive the data collected by the sensors. Given the fact that the sensors are not of the best quality and always provide inaccurate measurements, the player will try to deduce the monster's position and make a guess based on the limited information provided by the sensors.

More precisely, the system is composed of:

- A main server, called the *Broker*, that is up for connections from either the IoT sensors or the client. **In this project, you are to implement the *Broker*.**
- A client, called the *Subscriber*, that will interact with the player using terminal I/Os, send messages to the *Broker*, read its responses, and update its display accordingly. **You will be provided with the *Subscriber*.**
- NxN IoT sensors placed on every cell of the Grid. The sensors share the data they collect by connecting to the *Broker*. **The sensors will also be provided to you.**

1.2 The MQTT Broker

In this project, you will have to implement the MQTT *Broker*. The *Broker* implements a publish/subscribe message pattern, which makes it easy to provide a many-to-one or one-to-many message distribution system. More precisely, every IoT sensor will connect to the *Broker* to publish their data in a particular topic. Topics are a way to structure the Application Messages into different categories (e.g., sensors publish data in the `position` topic). On the other side, the *Subscriber* will connect to the *Broker* and subscribe to the topics it is interested in, in order to track the monster.

Both publishers (IoTs) and subscribers are MQTT clients. The publisher and subscriber labels refer to whether the client is currently publishing messages or subscribed to receive messages.

The broker is responsible for receiving all messages, filtering the messages, determining who is subscribed to each message, and sending the message to these subscribed clients. The broker also holds the session data of all clients that have persistent sessions, including subscriptions and testaments (see Sec. 2).

Even though the MQTT protocol is relatively simple, we will not be using all of its features in the context of this project. A recap of which MQTT features need to be implemented for this project is presented in Table 1. We will review in more details the MQTT protocol in Sec. 2.

	Implemented for this project	
Feature	YES	NO
Client Connection	X	
Persistent Session State	Bonus	
Security features		X
Subscription to topics	X	
Wildcards in topic filters		X
Publish messages	X	
QoS 0	X	
QoS 1 & 2		X
Keep Alive	X	
Retained Messages	Bonus	
Will and testament	Bonus	

Table 1: Recap on which MQTT features need to be implemented for this project

Operational Mode

When dealing with network connections, you can never assume that the other side will behave as you expect. It is thus your responsibility to check the validity of the messages you receive, and to check that the MQTT protocol is not violated. In this project, if the messages received from the *Subscriber* or the IoTs contain errors, are not well formatted, or if the MQTT protocol is violated in any way, the *Broker* is expected to close the network connection. The same goes for the *Subscriber* and the IoTs that will interrupt the communication if your *Broker* is not behaving as expected.

2 MQTT: The Standard for IoT messaging

"MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium."

Citation from the official MQTT 3.1.1 specification

The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections (in this project, we will be using TCP). Its features include:

- Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications,

- A messaging transport that is agnostic to the content of the payload,
- Three qualities of service for message delivery, providing so a reliable communication over unreliable networks,
- A small transport overhead and protocol exchanges minimized to reduce network traffic,
- A mechanism to notify interested parties when an abnormal disconnection occurs.

The MQTT protocol is an OASIS standard whose complete specification can be found here. However, as this specification is quite lengthy, we will briefly present in this statement the different aspects of MQTT to make things easier for you. For some aspects of the protocol, you will be asked to directly refer to the specification, which serves as a reference no matter what. Indeed, reading and conforming to specifications is one the most important jobs of people in the field of computer networking!

First, we'll explore the basic functionality (Connect, Subscribe, Publish, and Keep Alive) of MQTT. Then, we'll look at the more advanced features: Persistent Session, Retained Messages, and Last Will and Testament (for the bonus). You are also provided with an example packet capture (pcap) of the interactions between an MQTT *Broker* and its clients, in order for you to visualize clearly how the protocol is applied.

2.1 Data representation

In MQTT, different types of data can be encoded into a packet (integer values and strings, notably). The way those types of data are encoded is specified in the official MQTT 3.1.1 specification - Section 1.5

2.2 MQTT Control Packet format

An MQTT Control packet is always composed of a *fixed header* (FH), and sometimes of a *variable header* (VH) and of a *payload*, as can be seen in Table 2. A visual summary of the content of all MQTT packet types can be found in Sec. 2.10.

	bits	7	6	5	4	3	2	1	0
FH	byte 0	Packet type				Flags (specific to packet type)			
	byte(s) 1[...4]	Remaining length							
VH		Content (specific to packet type)							
Payload		Content(specific to packet type)							

Table 2: MQTT Control Packet Format

Fixed Header

The Fixed Header, present in all MQTT Control Packet, is composed of 2 to maximum 5 bytes. The first byte contains the packet type, as well as various flags specific to each MQTT Control Packet type. The flag values for each type of packet are presented in Table 4 and discussed (when relevant) in the next sections.

The rest of the Fixed Header is used to encode the Remaining Length, which is the number of bytes remaining within the current packet, including data in the Variable Header and the Payload. The Remaining Length does not include the bytes used to encode the Remaining Length. The Remaining Length is encoded using a variable length encoding scheme, meaning that the length of this field will vary based on the number it represents. Further explanations on how to encode and decode this field can be found in the official MQTT 3.1.1 specification - Section 2.2.3.

Variable Header and Payload

The Variable Header and the Payload will be described (when relevant) for each MQTT Control Packet.

2.3 Connection to the Broker

To initiate a connection, the client sends a CONNECT message to the broker (this packet must be the first to be sent after establishing the TCP connection).

- **Variable Header**: The Variable Header for the CONNECT Packet consists of four fields in the following order:
  - The protocol: "MQTT" (6 bytes).
  - The version of the protocol (3.3.1). The version 3.3.1 is encoded as the integer 4 (1 byte).
  - The connect flags (1 byte). The Connect Flags byte contains a number of parameters specifying the behavior of the MQTT connection. It also indicates the presence or absence of fields in the payload. There are six flags:
    - Username flag: specifies if the username is present in the payload.
    - Password flag: specifies if the password is present in the payload.
    - Will flag: specifies if a Last Will message is present in the payload (see Sec. 2.9)
    - Will retain flag: If the will flag is set, specifies if the Last Will message is a retained message or not (see Sec. 2.8)
    - Will QoS: If the will flag is set, specifies the QoS of the Last Will message.
    - Clean session: The clean session flag tells the broker whether the client wants to establish a persistent session or not. The clean session is discussed more in details in Sec. 2.7.

  - The keep-alive value (2 bytes). The keep-alive is discussed more in details in Sec. 2.6.

- **Payload**: The Payload of the CONNECT Packet contains one or more length-prefixed fields, whose presence is determined by their corresponding flags in the variable header. These fields, if present, must appear in the following order:

  - The client identifier. The client identifier identifies each MQTT client that connects to an MQTT broker. The broker uses the client ID to identify the client and the current state of the client. The client ID is unique per client and broker.
  - The will topic, if the will flag is set in the Variable Header. The will topic is discussed more in details in Sec. 2.9.
  - The will message, if the will flag is set in the Variable Header. The will message is discussed more in details in Sec. 2.9.
  - The username, if the username flag is set in the Variable Header.
  - The password, if the password flag is set in the Variable Header.

The broker responds with a CONNACK message and a status code. Once the connection is established, the broker keeps it open until the client sends a DISCONNECT command or the connection breaks.

Refer to the official MQTT 3.1.1 specification - Section 3.1 and Section 3.2 for a detailed explanation of the packets' content as well as their encoding.

2.4 Subscription to topics

In the publish/subscribe pattern, the publishers and subscribers never contact each other directly. In fact, they are not even aware that the other exists. The connection between them is handled by the *Broker*, whose job is to filter all incoming messages and distribute them correctly to subscribers. Clients can thus connect to the *Broker* and subscribe to the topics that interest them.

The SUBSCRIBE message is very simple, it contains a unique packet identifier and a list of topics the client wants to subscribe to. Each topic in the list is followed by the requested QoS for that topic.

To confirm each subscription, the broker sends a SUBACK acknowledgement message to the client. The SUBACK packet must have the same packet identifier as the SUBSCRIBE packet that it is acknowledging. The SUBACK packet must contain a return code for each topic/QoS pair. This return code must either show the maximum QoS that was granted for that subscription or indicate that the subscription failed (error code 128).

The UNSUBSCRIBE message is similar to the SUBSCRIBE message and has a packet identifier and a list of topics (without QoS). To confirm the UNSUBSCRIBE, the *Broker* sends an UNSUBACK acknowledgement message to the client (same packet identifier as the UNSUBSCRIBE packet).

Refer to the official MQTT 3.1.1 specification - Section 3.8, Section 3.9, Section 3.10, and Section 3.11 for a detailed explanation of the packets' content as well as their encoding.

Note on MQTT topics

MQTT topics are very lightweight. The client does not need to create the desired topic before they publish or subscribe to it. The *Broker* accepts each valid topic without any prior initialization.

In MQTT, topic names actually have a structure and can be divided into different levels. When a client subscribes to a topic, it can subscribe to the exact topic of a published message or it can use wildcards to subscribe to multiple topics simultaneously. For example, subscribing to the topic `myhome/groundfloor/#` will allow to receive all messages published in a topic that begins with the pattern before the wildcard character. In this case, the subscriber could receive messages in various topics such as:

- `myhome/groundfloor/livingroom/temperature`,
- `myhome/groundfloor/kitchen/temperature`,
- `myhome/groundfloor/kitchen/brightness`,
- etc.

However, for the sake of simplicity, we will not implement this wildcard subscription mechanism, and the *Broker* will only forward messages based on an exact-match of the topic name.

2.5 Publishing Application Messages

A PUBLISH Control Packet is sent from a Client to a Server or from Server to a Client to transport an Application Message. An MQTT client can publish messages as soon as it connects to a broker.

The client that initially publishes the message is only concerned about delivering the PUBLISH message to the broker. Once the broker receives the PUBLISH message, it is the responsibility of the broker to deliver the message to all subscribers. The publishing client does not get any feedback about whether anyone is interested in the published message or how many clients received the message from the broker.

To handle the challenges of a pub/sub system in an unreliable environment, MQTT has three Quality of Service (QoS) levels, which determines what kind of guarantee a message has for reaching the intended recipient (client or broker). There are three levels: level 0 (at most once delivery), level 1 (at least once delivery), and level 2 (exactly once delivery). In the context of this project, we will only implement the QoS 0, corresponding to a best-effort service. This level is also called the “fire and forget” level: if the subscriber is connected, it will get the message, otherwise the message will be discarded.

- **Fixed Header**: In a PUBLISH message, MQTT will finally make use of the flags defined in the Fixed Header (see Sec. 2.2). The Fixed Header of a PUBLISH message defines 3 fields, as can be seen in Table 3.
  - The dup flag: if set to 1, it indicates that this might be re-delivery of an earlier attempt to send the packet. This field is related to superior levels of QoS (>0), and is therefore of no concern to us in the context of this project, as we will only implement the QoS 0.
  - The QoS: this field indicates the level of assurance for delivery of an Application Message. In the context of this project, it will always be set to 0, as we only implement the QoS 0.
  - The retain flag: if set to 1, indicates that this message is a retained message (see Sec. 2.8).

	bits	7	6	5	4	3	2	1	0
FH	byte 0	Packet type				dup	QoS		retain
	byte(s) 1[...4]	Remaining length							

Table 3: PUBLISH packet Fixed Header

- **Variable Header**: Each message must contain a topic that the broker can use to forward the message to interested clients.

- **Payload**: Typically, each message has a payload which contains the data to transmit in byte format. Remember that MQTT is data-agnostic. The use case of the client will determine how the payload is structured (text data for example, or even full-fledged XML or JSON).

Refer to the official MQTT 3.1.1 specification - Section 3.3 for a detailed explanation of the packet's content as well as its encoding.

2.6 Keep Alive

When dealing with network connections, you can never assume that the other side will behave as you expect. It is the server's responsibility to ensure that a malevolent person won't waste resources by initiating a TCP connection and keep it open for an indefinite period of time.

As a counter-measure, the Keep Alive is a time interval measured in seconds, which represents the maximum time interval that is permitted to elapse between the point at which the Client finishes transmitting one Control Packet and the point it starts sending the next. If the Keep Alive value is non-zero and the Server does not receive a Control Packet from the Client within one and a half times the Keep Alive time period, it MUST disconnect the Network Connection to the Client as if the network had failed.

2.7 Persistent Session (Bonus)

If the connection between the client and broker is interrupted during a non-persistent session, the topics the client has subscribed to are lost and the client needs to subscribe again on reconnect. Re-subscribing every time the connection is interrupted can be a burden for constrained clients with limited resources. To avoid this problem, the client can request a persistent session when it connects to the broker. Persistent sessions save all information that is relevant for the client on the broker. The client ID that the client provides when it establishes connection to the broker identifies the session.

In a persistent session (flag CleanSession = false), the broker stores all subscriptions for the client, as well as the existence of a session. If the session is not persistent (flag CleanSession = true), the broker does not store anything for the client and purges all information from any previous persistent session.

2.8 Retained Messages (Bonus)

A retained message is a normal MQTT message with the retained flag set to true. The broker stores the last retained message and the corresponding QoS for that topic. Each client that subscribes to a topic that matches the topic of the retained message receives the retained message immediately after they subscribe. The broker stores only one retained message per topic. Often, it is not even necessary to delete a retained message, because each new retained message overwrites the previous one.

Retained messages help newly-subscribed clients get a status update immediately after they subscribe to a topic (without waiting until a publishing client sends the next message). Using retained messages helps provide the last good value to a connecting client.

In the context of this project, the IoTs will send a retained message as soon as they connect to the *Broker* in the `iot_x/status` topic to say they are online.

2.9 Last Will and Testament (Bonus)

Because MQTT is often used in scenarios that include unreliable networks, it is reasonable to assume that some of the MQTT clients in these scenarios will occasionally disconnect ungracefully. An ungraceful disconnect can occur due to loss of connection, empty batteries, or many other reasons.

The Last Will and Testament (LWT) is a mechanism to notify interested parties when an abnormal disconnection occurs. During a connection to the *Broker*, a client can include a LWT message in the packet. The broker will then be responsible for publishing the LWT of a client in the following situations:

- The broker detects an I/O error or network failure.
- The client fails to communicate within the defined Keep Alive period.
- The client does not send a DISCONNECT packet before it closes the TCP connection.

Control Packet	**Header**		**Variable Header**	**Payload**
	Type	Flags		
CONNECT	1	Reserved (0000)	protocol version connect flags keep alive	client ID (required) will topic (optional) will message (optional) username (optional) pwd (optional)
CONNACK	2	Reserved (0000)	SP flag & Return Code	/
PUBLISH	3	dup, QoS, retain	topic name	application message
SUBSCRIBE	8	Reserved (0010)	packet ID	list of topics (+QoS)
SUBACK	9	Reserved (0000)	packet ID	list of QoS
UNSUBSCRIBE	10	Reserved (0010)	packet ID	list of topics
UNSUBACK	11	Reserved (0000)	packet ID	/
PINGREQ	12	Reserved (0000)	/	/
PINGRESP	13	Reserved (0000)	/	/
DISCONNECT	14	Reserved (0000)	/	/

Table 4: MQTT Control Packet Recap

- The broker closes the TCP connection because of a protocol error.

In the context of this project, each IoT includes a LWT (in the `iot_x/status` topic) during its connection to the *Broker*, in order to notify interested parties it is offline in the event of an unexpected disconnection from the *Broker*.

2.10 MQTT Control Packets Recap

Table 4 provides a visual summary of the various MQTT packets' content. Refer to the official MQTT 3.1.1 specification for a detailed explanation of the packets' content as well as their encoding.

3 Launching the system

3.1 The Broker, the Subscriber, and the IoTs

The *Broker* must listen on port 2*xxx* – where *xxx* are the last three digits of your ULiège ID. Your program is to be invoked on the command line, with one additional argument for the port number the server should be listening to:

```
java Broker <2xxx>
```

The *Subscriber* as well as the IoT sensors are invoked on the command line like this:
`java -jar Subscriber_and_IoTs.jar <2xxx> <N>`
where `<2xxx>` is the port number of the *Broker*, and `<N>` determines the size of the NxN Grid.

3.2 The Cheating Mode

In order to make development and testing easier for you, the IoT devices can be launched in cheating mode by typing:
`java -jar Subscriber_and_IoTs.jar <2xxx> <N> cheat`

The cheating mode allows you to know at each iteration the path taken by the monster on the Grid, as well as the messages you will receive from the sensors, in order to verify that your implementation is consistent.

3.3 The IoT status checker (Bonus)

Additionally, you are also provided with a program, the `IoT status checker`, that will allow you to test the more advanced features of your *Broker*, namely, the Persistent Session, the Retained Messages, and the Last Will and Testament. It can be invoked on the command line with:
`java -jar IoT_status_checker.jar <2xxx> <N>`
where `<2xxx>` is the port number of the *Broker*, and `<N>` determines the size of the NxN Grid.

The IoT status checker is a program that will connect to your *Broker* with a Persistent Session and subscribe to each `iot_x/status` topic. If the retained messages have been implemented correctly in your *Broker*, the `IoT status checker` is supposed to receive the PUBLISH message from each IoT (saying they are online), even though they published it to the *Broker* when the `IoT status checker` was not yet connected.

After that, the `IoT status checker` will propose you to either reload the IoTs' status or to exit the program. If you choose the reload option, it will disconnect from the *Broker* and re-connect right after with a Persistent Session, but without subscribing to the IoTs topics this time. If the Persistent Session has been correctly implemented in your *Broker*, the `IoT status checker` is supposed to receive the retained messages from each IoT once again, without the need to re-subscribe to the topics.

Finally, in order to test the Last Will and Testament feature of your *Broker*, the `Subscriber_and_IoTs` program allows you to crash IoTs at will. Instead of entering your next guess into the program, you can enter `crash <tile>` (for example, `crash B3`) in order to crash the corresponding IoT. If you then reload the IoTs status with the `IoT status checker`, you should see a PUBLISH message from this IoT saying that it is offline.

4 Multi-threading and asynchronous behavior

4.1 A server's typical architecture

When a server accepts a connection, it usually invokes a new thread that will handle that connection, such that the server can go back to listening to the port for new incoming connections. This is very convenient to guarantee a certain level of accessibility but also has a flaw.

The *(Distributed) Denial of Service* (or (D)Dos) is an attack that targets servers with this kind of behavior. In this attack, one (DoS) or several (DDoS) machines initiate many bogus connections. If the server launches a new thread for each of these connections, it will soon encounter performance problems or even crash.

To circumvent this problem, one can use a *Thread Pool* that limits the number of threads that can be executed concurrently, while keeping the other jobs on hold until new threads become available. This thread pool can be implemented through the use of `java.util.concurrent.Executors` by calling the `newFixedThreadPool(int maxThreads)` method to create a fixed-size pool of *maxThreads* threads and calling the `execute(Runnable worker)` method to assign the work represented by `worker` to one of a thread in the pool, when available. You can (and are encouraged to) use a thread pool in your assignment.

4.2 An MQTT Broker

An MQTT *Broker* is quite different from a simple HTTP Server, that can deal with its clients relatively independently from each others. The job of the MQTT *Broker* on the other hand is to filter all incoming messages from its clients and distribute them correctly to subscribers. Clients can also be both *subscribers* and *publishers* at the same time. Basically, everyone can send anything at any time to an MQTT Broker.

As such, an MQTT *Broker* needs to work asynchronously: tasks cannot be blocked while waiting for or publishing a message. The *Broker* cannot stop listening to a client because it needs to publish a message to that client, or inversely, the *Broker* cannot delay the delivery of a message to a client because said client is also publishing something. In the same way, the *Broker* cannot stop listening/writing to a client because it needs to listen/write to another client.

To achieve such an architecture, you are advised to actually create *two* threads per client, a *listening* thread, and a *writing* thread. This paradigm will allow your server to be able to listen and transmit to a client at the same time. You also need to find a mechanism for coordinating those two threads, so that the server can easily answer a client in the writing thread based on what it received in the reading thread.

Furthermore, as the publications of certain clients will affect other clients, you also need to find a simple yet efficient way of synchronizing different clients' threads together. In other words, a PUBLISH message received in one of the clients' reading thread must easily be reflected in the (possibly multiple) subscribers' writing threads. You can have a look at `java.util.concurrent.BlockingQueue`, which is a thread-safe implementation of a queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element (if there is no more room).

5 Guidelines

- Your commented source code will be delivered no later than 17th of December 2021 (**Hard deadline**) to the Montefiore Submission platform (`http://submit.run.montefiore.ulg.ac.be/`) as a .zip package.
- You will implement the programs using Java 1.8, with packages `java.lang`, `java.io`, `java.net` and `java.util`,
- You will ensure that your program can be terminated at any time simply using CTRL+C, and avoid the use of ShutdownHooks
- You will not manipulate any file on the local file system.
- You will ensure your main class is named Broker, located in Broker.java at the root of the archive, and does not contain any `package` instruction.
- You will not cluster your program in packages or directories. All java files should be found in the same directory.
- You will ensure that your program is fully operational (i.e. compilation and execution) on the student's machines in the lab (ms8xx.montefiore.ulg.ac.be).

Submissions that do not observe these guidelines could be ignored during evaluation.

Your program will be completed with a report called report.pdf (in the zip package) addressing the following points:

- Software architecture: How have you broken down the problem to come to the solution? Name the major classes and methods responsible for requests processing.
- Reflection: The MQTT protocol provides three qualities of service for message delivery (level 0: At most once delivery, level 1: At least once delivery, and level 2: Exactly once delivery). However, we also know that MQTT runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. Therefore, what is the purpose of implementing QoS features in MQTT? What does it provide that TCP doesn't already provide?
- Limits & Possible Improvements: Describe the limits of your program, especially in terms of robustness, and what you could have done better.