Assignment Brief

Subject	LAS3006 – Modern Java for the Java Developer
Assessment	100% of subject marking
Deadline	Friday 13th January 2017 – 23:00 CET

Read the instructions carefully, thoroughly and completely before attempting this assignment. Failure to comply with the requirements can result in unnecessary loss of marks or potential disqualification.

Description

Write a lightweight highly scalable "Internet of Things" machine-to-machine message broker.

This broker must be able to scale up to thousands of *continuously connected* clients with minimal overhead. Clients, representing devices, can "publish" messages on a "topic". Other clients, representing customer applications, can "subscribe" to topics and get any messages published on these topics.

For example, your Fridge might publish its temperature on a topic:

`home/refrigerator/temperature`

While your bedroom Air Conditioning system might publish the room temperature on a topic:

`home/bedroom/temperature`

Each client will have its own unique ID (a string).

The broker should support the following messages:

Sent from Client	Sent from Server (Broker)	Initiated By	Description
CONNECT	CONNACK	Client	Client wants to connect. Its ID is provided in this initial message.
SUBSCRIBE	SUBACK	Client	Client wants to subscribe to a topic.
PINGREQ	PINGRESP	Client	Client wants to ensure link is still alive.
PUBLISH	PUBACK	Client	Client wants to publish on a topic.
PUBREC	PUBLISH	Server	Server wants to notify a (published) message to the client.
UNSUBSCRIBE	UNSUBACK	Client	Client wants to unsubscribe from a topic.
DISCONNECT		Client	Client wants to disconnect (gracefully).

You are NOT required to implement any persistence, guaranteed delivery or quality of service. When a message is published on a **topic** by a **client**, the broker must determine which other clients are currently subscribed to that topic and deliver the message to them. You are **NOT** required to implement any security. Refer to **Appendix A** for a suggested protocol you can use for these messages.

Topic Subscriptions

Topics are organised in a hierarchy. Each level is separated by a / character.

Each topic must have at least one character, and at least one level. Each level can be any string made up of alphabet characters, digits and punctuation **excluding spaces and the following characters**:

`/ # +`

A client can subscribe to a single specific topic, such as **`/home/bedroom/temperature`**

A client can also subscribe to a range of topics by using wildcards. The wildcards that need to be supported are as follows:

Single Level Wildcard: +

For example, **`home/+/temperature`** corresponds to a subscription to all topics which have 3 levels, with the first level being **`home`** and the last level being **`temperature`**

Messages published on the following topics will match such a wildcard subscription:

`home/refrigerator/temperature`
`home/bedroom/temperature`

While messages published on the following topic will **NOT** match such a wildcard subscription:

`home/bathroom/water-boiler/temperature`

Multi Level Wildcard:

For example, **`home/kitchen/#`** corresponds to all the topics which have the first 2 levels being **home** and the second level being **kitchen**. **Note that the # wildcard can only appear at the end.**

Messages published on the following topics will match such a wildcard subscription:

`/home/kitchen/temperature`
`/home/kitchen/humidity`
`/home/kitchen/refrigerator/temperature`
`/home/kitchen/freezer/temperature`
`/home/kitchen/gasdetector-status`

While messages published on the following topic will **NOT** match such a wildcard subscription:

`home/refrigerator/temperature`

Topic wildcards can be used in both SUBSCRIBE and UNSUBSCRIBE requests. However, if a wildcard topic is used in a SUBSCRIBE, the same exact wildcard topic string has to be used to UNSUBSCRIBE. (For instance, a client **cannot** SUBSCRIBE to **`/home/#`** and then UNSUBSCRIBE from **`/home/bedroom/temperature`**.)

A topic wildcard **cannot** be used with a PUBLISH message.

Implementation Requirements

The purpose of this assignment is to demonstrate practical applicability of the techniques covered during the course, namely:

- Lambda Expressions
- The Stream API
- Concurrency
- NIO
- Monitoring using JMX

Make sure that your implementation:

- Makes use of NIO to scale up easily without having to spawn a new thread per client connection.
- Makes use of a thread pool and/or other concurrency constructs.
- Uses the Stream API to perform any bulk operations on any collections used.
- Uses lambda expressions where deferred execution is needed.
- Offers a monitoring interface using JMX.

Functional Requirements

1. The broker must accept connections from clients and respond accordingly to CONNECT requests. Any malformed or out of sequence messages (such as a SUBSCRIBE or PUBLISH before a CONNECT) should be handled accordingly.
2. The broker must accept SUBSCRIBE requests and register the client's interest in that specific topic accordingly. It must also accept topic wildcards.
3. The broker must accept PUBLISH requests and process them accordingly, acknowledging their receipt to the publishing client and forwarding the published message to any clients interested in that specific topic.
4. The broker must accept UNSUBSCRIBE requests and unregister the client's interest in that topic accordingly. It must also support topic wildcards. It must reply accordingly whether the unregister request was successful.
5. Topics should be created on the fly, whenever a client subscribes to a topic. If a client publishes on an inexistent topic the message should just be discarded, since no one is interested. Be careful of wild-card topics.
6. The broker must monitor the activity of clients and if they are idle for X minutes (configurable) it must disconnect them. Clients can send a PINGREQ to keep the connection alive, to which the broker must reply with a PINGRESP and reset the timeout counter.
7. The broker must provide the right monitoring and instrumentation facilities through JMX. It should provide basic telemetry information such as number of clients connected, list of registered topics, number of messages delivered per topic, total number of messages delivered, number of messages delivered to each client, number of messages published by each client etc.

Dependencies

You should not use any third party APIs for this project, and you should use exclusively Java 8 SE and its APIs. The purpose is to demonstrate your full understanding to use these APIs.

Deliverables

You need to submit the code, together with any tests, and demo client applications, as Java source files. Include a **Maven** build file (pom.xml) which builds the various modules of the project, including the broker and any demo clients, and contains the common settings. Do not include any IDE specific project files. **Make sure no paths are hardcoded to your PC's directory structure**. Remember to set both your **source and target Java version to 1.8**. You have to set these in your pom.xml through the configuration settings of the inbuilt **`maven-compiler-plugin`**.

Your project should have at least the following modules (each will have its own child pom.xml):

1. A module for the broker.
2. A module for a test client simulating one or more devices, which publishes messages every few seconds.
3. A module for a test client simulating a subscriber interested in receiving information about the various devices that receives and in some way outputs or visualises the messages received.

Feel free to split the code into more modules if necessary. (For example you might want to make a common protocol messages library which both the broker and the clients use.). Include a readme.txt file at the root directory to indicate how to configure and use the executable classes. Add an **`exec`** goal to each module's pom.xml for easy execution of each executable class. You can use the **`exec-maven-plugin`** for this. Make the topics configurable, such that it is easy to change and test the clients with different topics. By default configure the test clients to connect to localhost (127.0.0.1).

Write-up

Accompany your program with a brief report of up to 3,000 words. You should describe the architecture of your program, a description of the main classes, the main design decisions you took, and how you verified that your program works as expected. Put this report as a pdf with the code.

Plagiarism

This assignment should be completed individually. **It is not a group assignment**.

Plagiarism is taken very seriously and any suspicion of plagiarism, copying, or claims of the work of others to be one's own will be thoroughly investigated and if confirmed, severe action will be taken according to the regulations of the University of Malta. This is not a group assignment, and each student is expected to carry out the tasks outlined in this assignment on his/her own.

For more details you can consult the university guidelines at:

https://www.um.edu.mt/__data/assets/pdf_file/0009/109476/University_Guidelines_on_Plagiarism_2010.pdf

A. Protocol

You are free to use any TCP/IP-based protocol of your choice. This project was inspired by the MQTT protocol (a real lightweight protocol designed for Machine-to-Machine communication) but you are **NOT** required to use MQTT. Especially since we will not be implementing persistence, quality of service or delivery guarantees. Also MQTT was designed to be an extremely lightweight binary protocol. In your case you can implement the messages as simple text messages, with each part of the message separated by a new line, and with a message ending with 2 consecutive new line characters.

For example:

```
CONNECT: 12345
```

Client with ID 12345 wants to connect.

```
CONNACK: 12345
RESULT: OK
```

The broker acknowledged the connection request from the client.

```
SUBSCRIBE: home/bedroom/temperature
```

The client wants to subscribe to topic **home/bedroom/temperature**

```
SUBACK: home/bedroom/temperature
RESULT: OK
```

The broker allowed the client to subscribe to the topic **home/bedroom/temperature**.

```
PUBLISH: home/bedroom/temperature
MESSAGE: 2142
PAYLOAD: temperature:12.5
```

The client wants to publish a message with ID 2142 on topic **home/bedroom/temperature**. The data payload is temperature:12.5

```
PUBACK: home/bedroom/temperature
MESSAGE: 2142
RESULT: OK
```

The broker acknowledged the publish request on topic **home/bedroom/temperature** of message 2142.

```
PUBLISH: home/bedroom/temperature
CLIENT: 12345
MESSAGE: 2142
PAYLOAD: temperature:12.5
```

The broker is broadcasting message 2142 published by client 12345 to the subscribed clients.

```
PUBREC: home/bedroom/temperature
CLIENT: 12345
MESSAGE: 2142
RESULT: OK
```

The client is acknowledging the receipt of message 2142 published from client 12345.

```
UNSUBSCRIBE: home/bedroom/temperature
```

The client wants to unsubscribe from topic **/home/bedroom/temperature** since it is no longer interested in receiving messages sent to that topic.

```
UNSUBACK: home/bedroom/temperature
RESULT: OK
```

The broker acknowledged the removal of the client's subscription to the topic.

```
PINGREQ: 11122
```

A client is informing the broker that the connection is still required and the broker is requested to acknowledge the request. The 11122 is a message ID.

```
PINGRESP: 11122
```

The broker acknowledged the client's request to keep the connection alive. Any connection timeout timers will be reset.

```
DISCONNECT
```

The client wants to disconnect. The broker will clean up any resources associated with the client, including any registered subscriptions, and terminate the connection.

This is just a suggestion. Feel free to adapt this protocol or use your own protocol if you prefer. It is suggested that messages carry some kind of message ID, as in the examples shown, with which the response can be correlated. This is useful to support asynchronous and out of order message acknowledgement if needed.