Exercise Sheet 4

Deadline: July 11, 2022 – 04:00 am CEST

1 MQTT Chat

In the previous exercises you have implemented a chat client using either a proprietary application layer protocol on top of UDP or using REST services. Both variants had some drawbacks when you consider the standard requirements of a chat application: typically you do not want to actively poll for received messages, but get automatically notified and probably you would like to have some authentication mechnanism in place. In this exercise, you are asked to implement another chat client based on the MQTT protocol to overcome these shortcomings.

1.1 Getting to know MQTT

Before you start implementing, you should first inform yourself about MQTT. The MQTT library of the Eclipse *Paho* project will be used for the implementation of the client application

MQTT is a *publish-subscribe* protocol that allows the transmission of data in the form of messages. A central *broker* establishes the connection to even resource-poor nodes. The messages are published under *topics*.

- The blog posts at `https://www.hivemq.com/tags/mqtt-essentials/` present a good introduction into the topic.
- For concise documentation you can also refer to the manpage from the *mosquitto* project: `https://mosquitto.org/man/mqtt-7.html`
- Familiarize yourself with the API documentation of the Paho-C API. `https://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/`
- There are examples listed in the documentation. Try these out.
- For the encryption and authentication aspects, look at the API documentation for the `MQTTClient_SSLOptions` structure and how it is used in the test programs of the Paho project (`https://github.com/eclipse/paho.mqtt.c/`).

1.2 Test with the mosquitto clients

For Linux there is the package *mosquitto* which provides a MQTT broker as well as ready-to-use command line tools for manual 'publish' and 'subscribe'. In case the mosquitto package is not available on your machine, you can a find statically linked version for Linux in the `mosquitto/bin` folder of your repository.

Here we will first test the function of the publish-subscribe protocol using the `mosquitto` tools.

There is a MQTT broker already configured and running at `mqttchat.dahahm.de`. However, you can also locally create your own `mosquitto` broker [1]. The configuration of the broker which you can also use for your own broker, can be found in the `mosquitto/configs` folder in your repository. (*note:* Attention if necessary paths have to be adapted!)

The mosquitto package provides generic publish and subscribe applications. These should be installed. With `mosquitto_sub` you can subscribe to arbitrary topics and output the messages to the console, and with `mosquitto_pub` to send messages to topics.

The MQTT broker is configured so that only certain topics can be created and used. can be used. However, for initial testing, there is the topic „`/test`“.

Subscribe to the topic with

```
mosquitto_sub -u <USER> -P <PASSWORD> -t "/test" -h
mqttchat.dahahm.de
```

In a second console, you can then send messages to the Topic with:

```
mosquitto_pub -u <USER> -P <PASSWORD> -t "/test" -h
mqttchat.dahahm.de -m "MESSAGE"
```

Next, start setting up your own small C program, that subscribes to the test topic. To make your program find the libraries of the Eclipse Paho project, before starting it, you have to source the file `env.sh` (command: `source env.sh`).

Send again with `mosquitto_pub` to the test topic and check their reception with your own subscriber program.

1.3 Exercise: Implementing the MQTT chat client

Each client needs to authenticate itself upon connection. By default you can authenticate yourself by using their lastname as username and password. (The same string that is the suffix of your git repository. E. g., if your git repository is `distributed-systems-ss22-foobar`, then your username and password will be `foobar`.) Once the client application starts the user shall login (and publish the corresponding message). The name of all active users shall be printed on the screen. Upon program termination the client shall perform a logout.

[1] projectpage: `https://mosquitto.org`, documentation at `https://mosquitto.org/man/mosquitto-8.html`

Furthermore, the client application shall provide the following methods

- send a message to a particular user
- send a message to a specific group
- join or leave a specific group for message reception
- change the user status

All received messages shall be printed on the screen immediately.

- **Topic:** `/chat/login`
  - **Description:** Each user should publish their name here upon starting the chat application.
  - **ACL:** user-rw, admin-rw
  - **QoS:** A subscriber should receive the message immediately exactly once.
- **Topic:** `/chat/logout`
  - **Description:** Each user should publish their name here before terminating the chat application.
  - **ACL:** user-rw, admin-rw
  - **QoS:** A subscriber should receive the message immediately exactly once.
- **Topic:** `/chat/active`
  - **Description:** Each user should publish their name here whenever a new user gets online.
  - **ACL:** user-rw, admin-rw
  - **QoS:** Confirmed message, at least once.
- **Topic:** `/chat/user/{USERNAME}/status`
  - **Description:** A user should publish their status upon change.
    States can be
    - ONLINE
    - AWAY
    - DO NOT DISTURB
  - **ACL:** user-rw, admin-rw
  - **QoS:** Best effort.
- **Topic:** `/chat/user/{RECEIVER USERNAME}/message/{SENDER USERNAME}`
  - **Description:** Can be used to publish a message for a specific user.
  - **ACL:** user-rw, other-w, admin-rw
  - **QoS:** Confirmed message, exactly once.
- **Topic:** `/chat/group/{GROUPNAME}/message/{SENDER USERNAME}`
  - **Description:** Can be used to publish a group message for an group.
  - **ACL:** other-rw, admin-rw
  - **QoS:** Confirmed message, at least once.

1.4 Exercise: Encrypted Communication via TLS

Besides the unencrypted connection to the MQTT broker (std. port 1883) there is also the possibility to connect encrypted via TLS (std. port 8883). To ensure that the client knows that it is dealing with the correct broker, a PKI (public key infrastructure) based on X.509 certificates is used. A CA (Certificate Authority) has signed the certificate signed by the broker. If you trust this CA, then you can determine whether the broker is also trustworthy.

Extend your program in such a way that an encrypted connection to the broker is (switch on via the `-v` program parameter flag). Use the provided certificate of the CA for the encrypted connection (`-v`). certificate (`mqtt_ca.crt`) for the encrypted connection to ensure that the connection is secure.

Use appropriate means to check the data communication. Are the messages transmitted encrypted?

Generate your own CA certificate and use it. Is it recognized that the server cannot be trusted?

1.5 Authentiation via X.509

Using X.509 certification, the client application can also authenticate itself to the server. For this purpose, our MQTT broker on port 8884 offers another access.

Use OpenSSL to create an RSA key pair for the IoT application with a minimum length of 2048 bits and a corresponding X.509 certificate. Fill in the necessary details in a meaningful way. Setting a password is optional.

Then generate (also with OpenSSL) a CSR (Certificate Signing Request) for your certificate and send it to `oliver.hahm@fb2.fra-uas.de`. You will receive a certificate signed with the private CA key. Import this into their key and use it for authentication.

Document their actions: Put their generated certificate files in the `certs` folder and add them to the git repo.

Extend your application to use your certificate for authentication to the broker. This should be done using the program parameter flag `-V` to enable this.