Protocol Development – Exercise

STOMP stands for **S**imple **T**ext **O**rientated **M**essaging **P**rotocol. It is effectively a very simple text-based protocol, designed for working with message-oriented middleware. It provides an interoperable wire format that allows **STOMP** clients to talk with any message broker supporting the protocol. The protocol is broadly similar to **HTTP** and works over **TCP**. The complete documentation/specification can be found here: https://stomp.github.io.

We chose **STOMP** as the protocol to implement in this exercise because it is very simple to understand and to implement on the client side.

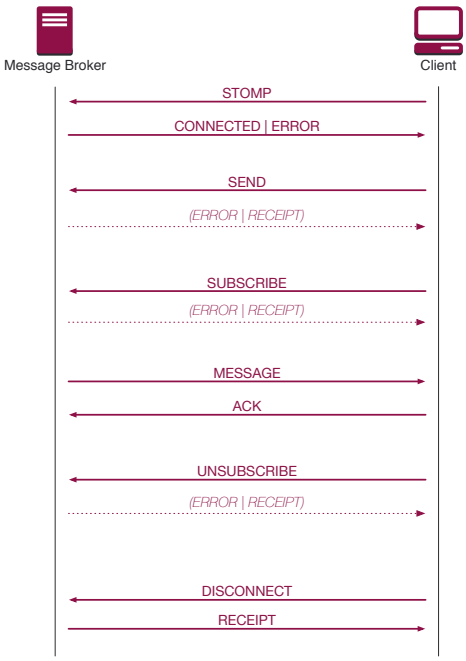

Figure 1: Typical STOMP message exchange.

The figure above illustrates a typical **STOMP** connection sequence. First a client has to open a TCP connection to the broker and immediately send the **STOMP** frame – This frame includes the username and password of the user that wants to connect. The broker will respond with a **CONNECTED** frame if the login was successful or with an **ERROR** frame otherwise.

Once connected, a client can publish a message at any time using the **SEND** frame. If the send frame requests a receipt and the message could be delivered, the broker responds with a **RECEIPT** message, if there was an error publishing the message, the broker will respond with an **ERROR** frame in any case.

A client can use the frames **SUBSCRIBE** and **UNSUBSCRIBE** to subscribe/unsubscribe from message destinations. Again, the broker responds with a **RECEIPT** frame on success if such a receipt is requested by the client and responds with an **ERROR** frame on failure.

Whenever another client publishes a message to a destination the client has subscribed before, the broker delivers the message using the **MESSAGE** frame. A client has to acknowledge the message using the **ACK** frame in any case (Or refuse it using the **NACK** frame).

A client can close the connection to the server by sending the **DISCONNECT** message and waiting for the **RECEIPT** message from the server.

STOMP offers additional functionality like Heartbeat (Keep-Alive) and Transactions which will not be used in the context of this exercise.

You are going to implement a basic **STOMP v1.2** client. The final goal will be to control a virtual vehicle in a very simple game called "Gem-Hunter" at https://gemhunter.sdi.hevs.ch using **STOMP** messages.

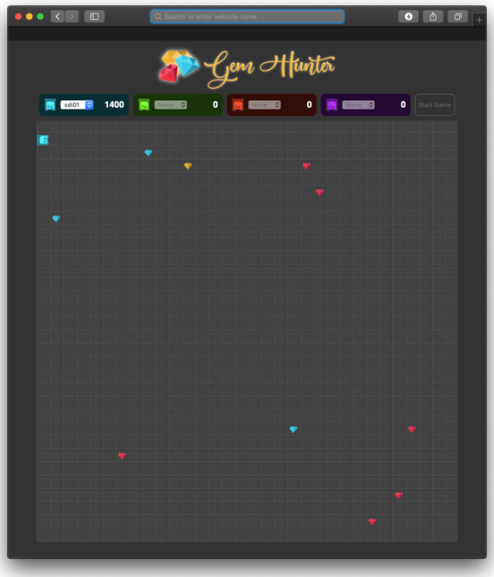

Figure 2: Gem Hunter web-based game.

The game will publish periodically (Every 500 ms) the game field to the destination(s):

/topic/sdi*XX*.gem.field

where *XX* is your group's number. The content of the message describes the actual state of the game field. The field is 32x32 tiles in size. The field is encoded as a single ASCII string, where each line of 32 characters represents a horizontal line on the game field and the string is exactly 32 lines long. Each object on the game field is represented by an ASCII-character:

' ' *(Whitespace)*	Empty field.
'g'	Gem worth 100 points.
'G'	Gem worth 250 points.
'D'	Diamond worth 500 points.
'Y'	Your own vehicle.
'h'	Other vehicles.

Figure 3: Game field ASCII characters and their meaning

Each vehicle on the game field can be controlled by sending messages to the destination:

/topic/sdi*XX*.gem.command

using the four simple commands **'up'**, **'down'**, **'left'** and **'right'** as the body of the published message.

1. GitHub classroom

1. Join the assignment for this exercise by visiting this link: https://classroom.github.com/a/AuzvVIQg.
2. Clone the repository to your local computer.

The project already contains these 4 files:

main.cpp	Minimal Qt main file.
stompframe.h	Header of **STOMPFrame** class. This class simplifies encoding/decoding of STOMP frames from/to any **QIODevice**.
stompframe.cpp	**STOMPFrame** class implementation.
STOMPClient.pro	**Qt** build instructions. **Qt Creator** will open the file as a project.

Open the project in **Qt Creator** using *File → Open File or Project…* and select the **STOMPClient.pro** file. Now you should be ready to work on the exercise.

2. Analysis & Design

1. Study the STOMP 1.2 specification.
2. Study the API of the QTcpSocket class which will be used as transport layer.
3. Design the protocol layers using UML diagrams according to the theory seen in the *Protocol Development* course.

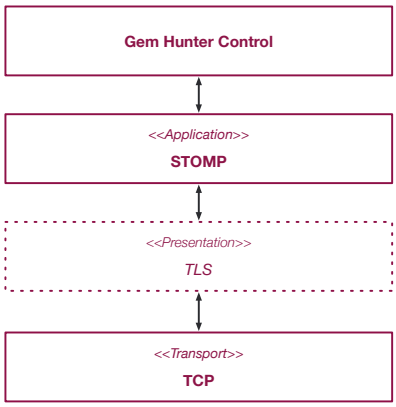

Figure 4: Protocol layers

The API of the transport layer (TCP) is given by the **QTcpSocket** class. If encryption is used, an additional layer (TLS) is introduced at the OSI Presentation layer and the **QSslSocket** class has to be used instead. Conceptionally the API of both classes are mostly identical, so you can model your solution using the **QTcpSocket** class.

You can freely choose the UML tool for this task; however, we ask you to export PDF files or images of your final diagrams to enable us to examine them.

4. Commit all your design files, tag your HEAD branch as **Design** and push your changes (including tags) to your assignment's GitHub repository.

3. Implement publishing

1. Implement your **STOMP client** design and test it by writing a simple application to control the vehicle in the Gem Hunter game manually (Application).

You can freely choose the kind of manual control you like to implement – This can be a console application that listens for keystrokes (WASD) or a GUI application with buttons to control the vehicle or anything else you could imagine.

In order to achieve this, your STOMP implementation must be able to:

- **Connect to the broker** using the transport layer (TCP & QTcpSocket or TLS and QSslSocket).
- Send the **STOMP frame** in order to authenticate and initiate the session.
- Handle the **CONNECTED** or **ERROR frame** from the broker as a response to the STOMP frame.
- Report the result of the connection attempt to the application.
- Send **a SEND frame** to the broker.
- Report any error to the application.
-

Access to the message broker

Depending if you work from inside the HEI network or from home, you need to connect to the SDi message broker differently.

Inside HEI network

You can simply use the **QTcpSocket** class to connect to the message broker without any encryption.

Outside HEI network (from home)

Allowing a non-encrypted connection from the internet to any service represents a non-tolerable security risk. Basically, everyone that can see the IP packets passing would know your username and password immediately. To avoid this, only TLS secured connections are allowed from outside the school's network.

This means that you have to use the class **QSslSocket** instead of **QTcpSocket**. The API is almost the same with some exceptions:

- As the message broker uses a **self-signed certificate**, you need to disable certificate verification using **setPeerVerifyMode(QSslSocket::VerifyNone)** before connecting, otherwise the connection attempt will fail.

- Use the method **connectToHostEncrypted()** instead of **connectToHost()**.

- Wait for signal **encrypted()** instead of **connected()** or use **waitForEncrypted()** instead of **waitForConnected()**.

Use these credentials to connect to the broker:

Host	sdi.hevs.ch
Port	61613 for unencrypted (QTcpSocket) 61614 for encrypted (QSslSocket)
vHost	/
User	sdi*XX* where *XX* is your group
Password	MD5 checksum of username

The password is the **MD5 checksum** of the user. You can visit http://www.md5generator.de in order to generate the MD5 checksum for your username.

2. Once your client implementation is able to control the vehicle in the game, commit your code and tag it as **Control** and then push the changes to your GitHub repository.

4. Implement subscribing & message consuming

3. Complete your **STOMP client** implementation and add support for subscribing to destinations and consuming messages received from the broker. You have two options in order to tests this functionality by extending your existing application:
   a) Your test application displays the game field locally. Either on the console using ASCII art or in a GUI window.
   b) Your application automatically controls the vehicle based on the received game field and collects as many gems as possible by sending the appropriate commands.
4. When finished, commit your code, tag it as **Completed** and then push the changes to your GitHub repository.