Institute of Information Systems Engineering

Distributed Systems Group (DSG)
VU Distributed Systems 2024W (194.024)

Assignment 1

Submission Deadline: 25.10.2024, 18:00 CEST

Contents

1 General Remarks

Please read the assignment carefully before you get started. Section 2 explains the application scenario and presents the specification of individual components. If something is not explicitly specified (e.g., the behavior of the application in a specific error case) you should make **reasonable assumptions** for your implementation that you can justify and discuss during the interviews.

1.1 Learning Objectives

In this assignment you will learn:

- the basics of TCP socket communication
- how to implement your own application-layer protocol(s)
- how to implement multi-threaded applications

1.2 Guidelines

1.2.1 Assignment Mode

This assignment is **individual** work only. Therefore, group work is not permitted. While we encourage you to exchange ideas and engage in discussions with your colleagues, the code you submit must be entirely your own.

1.2.2 Code Repository

The repository provided upon joining the GitHub Classroom assignment is set to **private** by default. **Do not** change the repository visibility to **public**. If you choose to use other version control hosting platforms, ensure that your repositories remain private.

1.2.3 Generative AI and Tools

The use of code generation tools is permitted, but you **must fully understand** both the code and the corresponding theoretical background of any code you submit.

1.3 Grading

The achievable points of this assignment are based on automated tests. Below is a brief overview of our grading process. Refer to Section 3 for more details.

- Push your code to the `main` branch of the assignments repository provided by GitHub Classroom.
- The last *Grading Workflow* score, counts towards your final assignment grade.
- Test your code locally using the provided JUnit test suite before pushing.
- No late submissions accepted, submit before the deadline.

If you have questions, the DSLab Handbook[1] and the TUWEL forums[2] are good places to start.

[1] `https://tuwel.tuwien.ac.at/mod/book/view.php?id=2388365`
[2] `https://tuwel.tuwien.ac.at/mod/forum/view.php?id=2388392`

2 Implementing a Message Broker Client

In this assignment, you will implement a client application that communicates with a message broker using the Simple Message Queuing Protocol (SMQP). This client will interact with the message broker via a TCP-based protocol, allowing for basic operations such as creating exchanges, publishing and subscribing to queues, and binding queues to exchanges. This assignment forms the foundation for understanding client-broker communication in a distributed system. Each assignment will build on top of the previous one. Figure 1 shows a simplified version of the final system.

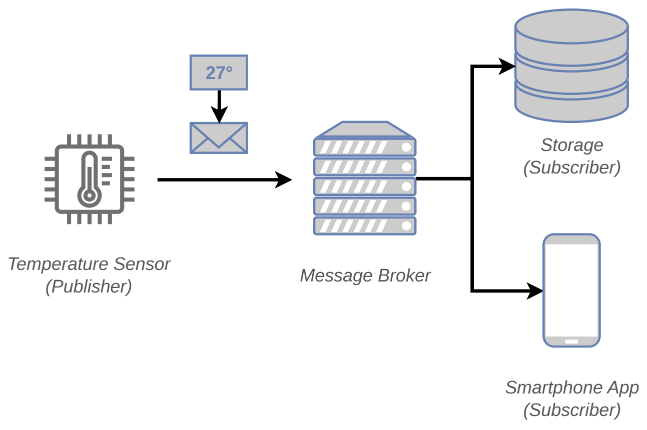

Figure 1: Message Broker - Bird's Eye View

2.1 Application Overview

The core components are:

- The **Client** is the main entry point of the application, through which users interact via the command-line interface. The Client allows users to easily communicate with the message broker, such as publishing or receiving messages.
- The **ClientCLI** is the component that reads user input from the command-line interface (i.e., `System.in`) and outputs responses to the command-line interface (i.e., `System.out`).
- The **Channel** handles communication with the message broker and follows the Simple Message Queuing Protocol. Figure 2 illustrates how the components interact with each other.

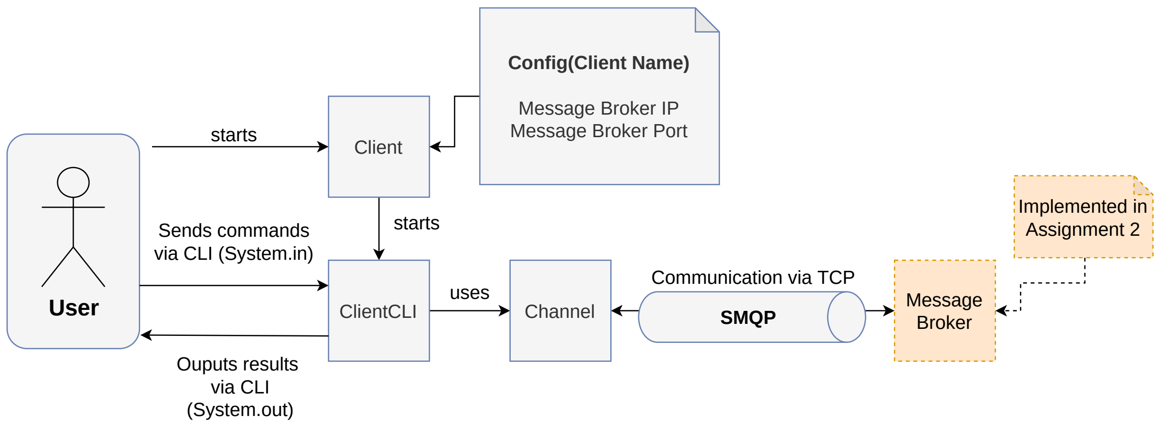

Figure 2: SMQP Component Interactions

2.2 Simple Messaging Queuing Protocol (SMQP)

The SMQP is a TCP-based plaintext application-layer protocol, that is used by both publishers (senders) and subscribers (receivers) to communicate with the message broker. The message broker is an intermediary to deliver messages to the correct queues, which clients can consume. Both the client and message broker implement SMQP, as illustrated in Figure 3.

2.2.1 SMQP Key Actions

- **Create Exchanges**: Define where the messages should be routed. Can be from any of the following types: *default*, *direct*, *fanout*, *topic*.
- **Create Queues**: Define queues that hold the messages for its subscribers. Each queue is associated with **exactly** one client.
- **Bind Queues to Exchanges**: Specify how queues should be associated with exchanges using a binding key.
- **Publish Messages**: Send messages to specific exchanges, with a routing key determining how the message should be routed.
- **Subscribe to Queues**: Listen for incoming messages from specific queues.

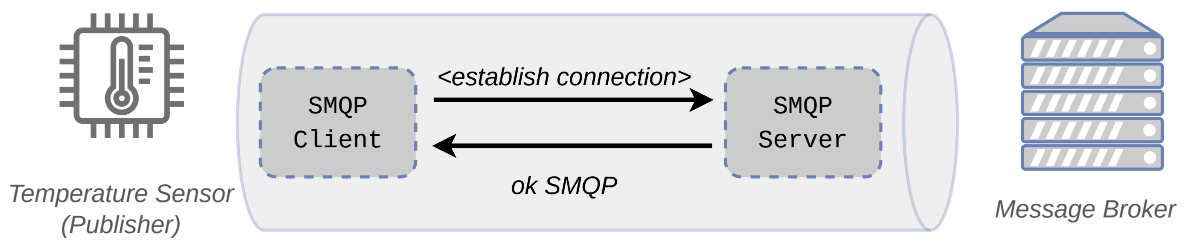

Figure 3: SMQP interaction

2.2.2 SMQP Commands

The client will use a sequence of the following commands **to communicate with the broker**.

- `exchange <type> <name>`: Create an exchange of the specified type (e.g., *fanout*) with the given name.
- `queue <name>`: Specifies the queue and creates it, if it does not exist.
- `bind <binding-key>`: Bind the previously specified queue to exchange with the specified binding key.
- `subscribe`: Subscribe to the created queue to receive messages.
- `publish <routing-key> <message>`: Publish a message to the exchange using the provided routing key.
- `exit`: Disconnect the client from the server.

Statefulness The SMQP commands need to be executed in the correct order and the message broker will keep track of the previous commands. This means that if a client sends *exchange fanout animals* to the message broker, the client **must** send the `queue` command afterwards and then the `bind` command.

After that, the client can either subscribe to the just created queue or publish to it. To create a new queue to publish/subscribe, the client first needs to `exit` and disconnect from the message broker.

Routing and Binding Key While binding a queue to an exchange is necessary, not all Exchange Types require a *Binding Key*. The same holds for *Routing Key*. For example, the exchage type *fanout* does not utilize any specified key. In such cases, we specify *none* as the routing/binding key to keep the implementation simple.

String Arguments You can assume that arguments do not contain any whitespace. For example publishing "maine coon" as message is not supported, but must be "maine-coon" instead. This holds for all string arguments, which means you can split the input of users from `System.in` by whitespace.

2.2.3 Message Broker Responses

A message broker that accepts SMQP instructions responds immediately to some commands (i.e., exchange, queue, bind, subscribe) while others may receive further additional responses at any point of time (i.e., subscribe). For commands that respond immediately, the message broker sends *"ok"* back to signal the client that the command was executed successfully. Each time an invalid or unknown command is received, the message broker responds with an error. The error response starts with *"error"*, and might contain further explanation. When a client connects, the message broker initially sends: *"ok SMQP"*, indicating that the message broker is ready and supports SMQP.

2.3 Client

You will implement the client-side logic to issue these commands and interact with the message broker, laying the groundwork for the next assignment where you will implement the message broker/server-side logic.

Client Application Details:

- The client will expose a command-line interface (CLI) for users to interact with the message broker.
- The client application will create a communication channel with the message broker over a TCP connection.
- Each client will connect to a single message broker, specified in the configuration file provided at client startup.
- The configuration file is named after the client name (Component ID), and contains message broker connection details (IP address and port) as key-value pairs.
- We provide a **Config** class that provides actions to read from the property file.
- Each client is limited to managing a single queue, which it can use for both publishing and subscribing.

Implementation: When communicating with the message broker (i.e., sending SMQP commands), you need to send the commands in the correct order to the connected TCP socket. In Assignment 2, you will implement the server-side logic (i.e., the message broker), which also adopt SMQP and therefore will be capable of processing the SMQP commands.

2.4 Client CLI

Some of these commands are similar to those in our protocol, but they abstract away underlying details to simplify user interaction with the message broker.

Note: the tests require you to implement `printPrompt` method.

2.4.1 Client CLI Commands

The client must validate the arguments of the following commands. A rule that applies to all of them is to check if all required arguments are provided; otherwise, print an *"error"* message. Section 2.4.2 describes each command in more detail, and the following provides an overview of available CLI commands with its expected arguments.

- `channel <broker-name>`: Create a channel to the specified broker.

  Note: <broker-name> is used to obtain the IP address and port from the configuration file.

- `subscribe <exchange-name> <exchange-type> <queue-name> <binding-key>`: Uses the channel to create a queue that binds it to the exchange to receive messages.

- `publish <exchange-name> <exchange-type> <routing-key> <message>`: Uses the channel to publish a message to the specified exchange using the provided routing key.

- `shutdown`: Disconnects the client from the message broker and terminates the client application.

2.4.2 CLI Commands Behavior

Channel:

- Holds connection to the message broker.
- Holds only one active Exchange.
- Holds only one active Queue.
- Prints an error message (i.e., *"error"*) in case the broker is not found (i.e., the broker connection details do not exist in the configuration file).
- Prints an error in case a connection to the broker is not possible (i.e., broker is offline / not reachable).
- Prints a success message (i.e., *"ok"*) if the channel has been created, in any other case print an error message.
- Repeatedly submitting the Channel command should disconnect the Client first from the Message Broker, and then re-establish a TCP connection to the Message Broker.

Subscribe:

- When a user executes the subscribe command and no error happens, the application prints out any incoming messages to `System.out` by starting a `Subscription` as a separate **background-thread**. After the background-thread has been started, the application waits for any input on `System.in`. Upon receiving any input (i.e., any arbitrary input is accepted), the background-thread is interrupted, stops the subscription, and returns to the applications main command loop.
- The CLI should only support one Subscription at a time.
- In case the subscribe command is executed without creating a channel first, an error message should be printed to the CLI.
- If any other error happens during the execution of the subscription process, an error message should be printed to the CLI.

Publish

- In case the publish command is executed without creating a channel first, an error message should be printed to the CLI.
- If any other error happens during the execution of the publishing process, an error message should be printed to the CLI.

2.5 Examples

The following examples are split in two categories.

- The first part illustrates the interaction between a Message Client and a Message Broker using SMQP. The Message Broker provided with the test suite is a simple, mocked server that operates in a "fanout" fashion. Despite the simple implementation of the mocked server, the Message Client must send all possible SMQP commands to pass the tests. These commands represent the internal communication between the Client and Broker and are not visible to users of the Client CLI.
- The second part illustrates the interactions between a user and the Client CLI. These are the commands that users of the Client CLI type in to interact with a Message Broker (i.e., via `System.in`).

These examples only serve as support for you to understand the message flow and you will be able to try them out when implementing Assingment 2.

2.5.1 Message Client to Message Broker via *SMQP*

Example 1 - Subscribe Here is a sample interaction between an SMQP client `C` and a broker `B`. In this example, `C` declares a *fanout* exchange named *animals*. After declaring the exchange, a queue *cats* is created and bound to the previously declared *animals* exchange. For the bind command, we pass *none* as the argument, since this argument is ignored for this type of exchange. Subsequently, the client subscribes and prints out any incoming messages via `System.out`. The Client listens to these messages in the **background** by starting a `Subscription` background-thread and interrupts it when detecting an input from the CLI's `System.in`. Afterwards the Client disconnects from the message broker by sending `exit`.

```
(C connects to B)
B: ok SMQP
C: exchange fanout animals
B: ok
C: queue cats
B: ok
C: bind none
B: ok
C: subscribe
S: ok
(the client blocks and prints out any messages that the message broker writes into the
    socket via System.out. This is done in the Subscription Background Thread.)
B: maine-coon
B: british-shorthair
(the client detects input from System.in and interrupts the Subscription)
C: exit
B: ok bye
```

Example 2 - Subscribe and Publish Here is a sample interaction between a SMQP Subscriber `S`, a publisher `P` and a message broker `B`. Refer to Figure 4, which provides a timeline visualization of the different component states and the SMQP commands transmitted in the background. First `S` connects to `B` and declares a *fanout* exchange named *animals*. After declaring the exchange, a queue *cats* is created and bound to the previously declared *animals* exchange. Next, `S` sends the SMQP subscribe command to `B`. After `B` has confirmed the successful processing of the command back to `S` (i.e., "ok" command), `S` starts a background thread (i.e., *Subscriber Thread*), which handles incoming messages from the subscribed queue. The main thread, in which `S` is running, will block and wait for any input from `System.in`. We can now publish messages to the Exchange, which the queue of `S` is subscribed to, by connecting `P` to `B` and sending the SMQP publish command sequence to `B`. After publishing two messages (just for this

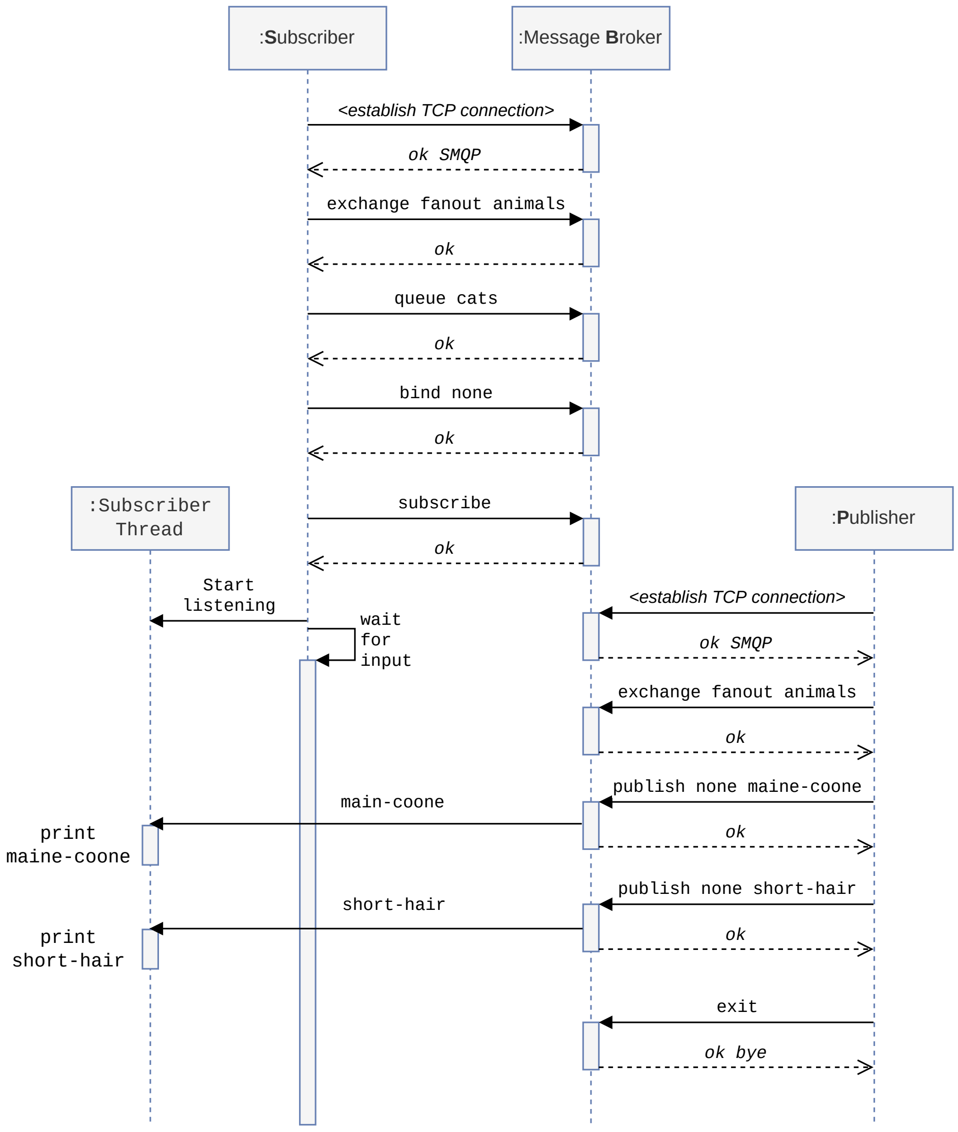

Figure 4: Sequence Diagram of Subscribe and Publish

example), client `P` disconnects from the message broker by sending the `exit` command. Moreover, `S` continues to wait for any input and the *Subscriber Thread* keeps waiting for new messages from `B`.

2.5.2 User and Client CLI via `System.in` and `System.out`

Example 3 - Client CLI subscribe In this example, the client creates a Channel, and then subscribes to a given Exchange. In this assignment your task is to only send the SMQP commands to the server and create a subscriber thread (Subscription).

```
C: channel broker-0
(C connects to B - broker-0)
B: ok SMQP
C: subscribe animals fanout cats none
(the client blocks and prints out any messages that the Server writes into the Socket via
     System.out. This is done in the Subscription)
B: maine-coon
B: british-shorthair
(the client detects input from System.in and interrupts the subscriber thread
    Subscription, then the client returns to the main loop of the program and prints the
    prompt)
```

```
C: shutdown
(C disconnects from B by sending the exit command internally)
B: ok bye
```

In the example above, when the client submits the *subscribe* command, your implementation should send the correct SMQP commands to the broker. Example 1 shows the process of subscribing.

Example 4 - Client CLI publish In this example, the client creates a Channel, and then publishes twice to a exchange named *animals* with "none" as routing key. In this assignment your task is to only send the SMQP commands to the server.

```
C: channel broker-0
(C connects to B - broker-0)
B: ok SMQP
C: publish animals fanout none maine-coon
B: ok
C: publish animals fanout none british-shorthair
B: ok
C: shutdown
(C disconnects from B by sending the exit command internally)
B: ok bye
```

In the example above, when the Client submits the *publish* command, your implementation should send the correct SMQP commands to the Broker. Example 2 shows the process of publishing.

Note: In Example 3 and Example 4, the input of client `C` is performed on the CLI and not on the channel between client and broker. The responses from the message broker `B` are related to the established channel connection between `C` and `B`.

2.6 Outlook

This assignment serves as a primer, ensuring that you understand the client's role in a message-driven architecture and prepares you for deeper exploration in the next upcoming assignments. For now, the client will not interact with a real message broker. Instead, it will focus on implementing the core functionalities of handling local TCP connections, implementing a own application-layer protocol, reading from the configuration file. Interaction with an actual message broker will come in Assignment 2, where the server-side component is introduced.

3 Automated Grading, Testing and Protected Files

This section offers a comprehensive guide on how to run the provided tests to verify the correctness of your assignment solution. It also gives an overview regards the grading process, including the criteria used for assessment. Additionally, it denotes "protected files and paths" that must remain unchanged, ensuring you avoid any point deductions due to modifications of restricted files.

3.1 Protected Files and Paths

Certain files and paths are "protected" and must stay untouched at any given time. Altering any of these will result in your repository receiving a permanent **non-removable** flag indicating unallowed changes, leading to point deductions. This restriction extends to creating, renaming, or deleting files or directories in these protected areas. The provided *.gitignore* file is designed to help prevent unintentional commits of unallowed changes.

3.1.1 Files and Paths you must leave unmodified

- .github/**/*
- src/main/resources/**
- src/test/**/*
- pom.xml

It is your responsibility to ensure that none of the protected files or paths are modified. In the worst-case scenario, unallowed changes may result in a final score of 0 points.

3.2 Grading Workflow with GitHub

Grading is conducted entirely through GitHub via the *Grading Workflow*. You are required to submit your code solution to the assignment's remote repository created by GitHub Classroom using `git push`. Only pushes to the `main` branch will initiate a *Grading Workflow*. **Important:** We cannot accept any late submissions (i.e., hard deadline), so ensure your solution is pushed before the assignment deadline.

To start a *Grading Workflow*, push your (partial) solution to the `main` branch of the above stated GitHub repository. Each unit test is marked with a `@GitHubClassroomGrading` annotation, indicating the score achievable if the test-case result is successful. The score from the last completed *Grading Workflow* is the one that will count toward the final assignment grade.

Do not worry if GitHub displays a red cross - "failed", after the *Grading Workflow* has finished. You will still earn points for any test-cases that passed in the *Grading Workflow*. If all tests pass, you will see a blue checkmark indicating "success". In this case you will earn the maximum **score of 40**, which is then **translated to 4 points** (i.e., dividing by 10). The translated assignment points will be uploaded to TUWEL after the assignment deadline has passed.

Grading Workflow Limitation: Only one *Grading Workflow* can run at a time per assignment repository (i.e., per student). If you push a new code version to your repository while a *Grading Workflow* is still in progress, the "still in progress" Workflow will be cancelled and a new *Grading Workflow* for the new version will be added to the end of the waiting queue. This could lead to a longer wait time, since you may be queued behind a lot of students already waiting in the queue.

3.3 Test Suite and Local Testing

The JUnit test suite used for the Grading Workflow with GitHub is identical to the one provided in the assignment template. Since the infrastructure hosting the *Grading Workflow* has limited execution capacity, we highly recommend testing your solution on your local machine before pushing any changes to the remote repository's *main* branch.

You can run the JUnit test suite locally to check your progress and estimate the points you may earn for the coding part. Running the entire test suite will generate a `‘‘Grading Simulation Report’’`, which is printed directly to the end of the output console.

Use the following commands to verify your solution:

- `mvn test` (runs the entire test suite)
- `mvn test -Dtest="<testClassName>#<testMethodName"` (runs a specific test method)

3.4 Summary

The following highlights the necessary steps to verify your solution:

1. Commit and push your code solution to the `main` branch of the repository provided by GitHub Classroom.
2. Wait for the Grading Workflow to complete.
3. Check the score and calculate the points (i.e., by dividing through 10).
4. **Do not modify test files or any others mentioned above.**

There is no dedicated submission interview for Assignment 1. However, the concepts you have learned here will be important for the following assignments and the submission interview of Assignment 2.

4 Submission

Commit and push your code solution to the `main` branch of the repository provided by GitHub Classroom. The score from the last completed *Grading Workflow* is the one that will count toward the final assignment grade.

4.1 Checklist

- ☐ Commit and push your solution ahead of the deadline.
- ☐ Check if the tests are passing as expected in the Grading Workflow.

4.2 Interviews

There is no dedicated submission interview for Assignment 1. However, the here learned concepts will be important for the upcoming assignments and the submission interview for Assignment 2.