Institute of Information Systems Engineering

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

Assignment 2

Submission Deadline: 29.11.2024, 18:00 CET

Last modified: 07.11.2024

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:

- How to implement a TCP server that communicates over sockets.
- How to manage application-layer protocols, specifically SMQP, in a multi-threaded environment.
- How to maintain state across multiple clients in a server environment.
- How to process SMQP commands from clients and respond with appropriate responses.
- How to register, unregister, and resolve domain names using the Simple DNS Protocol (SDP).

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. There will be no room for interpretation, nor will we make any exceptions if the use of generative tools results in plagiarism. Therefore, use these tools at your own risk.

1.3 Grading

The achievable points (20) of this assignment are based on automated tests and a submission interview. You can achieve a score of 100 (i.e., 10 points) if all tests pass, and an additional 10 points during the submission interview. Below is a brief overview, refer to Section 4 for more details.

- Push your code to the `main` branch of the assignments repository provided by GitHub Classroom.
- The last achieved *Grading Workflow* score, counts towards your final assignment grade. Scores from **re-runs** of the *Grading Workflow* are **not** considered for your final assignment grade.
- No late submissions accepted, submit before the deadline.

Submission Interview In the submission interview, you must be able to explain your solution and answer questions that assess your deeper understanding of the technologies used for implementing this assignment. We also manually check your submission for non-functional requirements that are not verified by any automated tests. The assignment description includes remarks referring to non-functional requirements (e.g., correct concurrency handling) that constitute a proper implementation.

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=2388410

2 Implementing a DNS Server and a Message Broker

In this assignment, you will implement two server applications, a DNS (Domain Name System) server and a Message Broker. The DNS server implements the TCP-based Simple DNS Protocol (SDP), which allows message brokers to register a domain, and clients to resolve a domain to an address. The Message Broker implements the server-side of the Simple Message Queuing Protocol (SMQP), with which you already interacted in Assignment 1 from the client-side.

By the end of this assignment, you will have implemented a DNS server and a Message Broker, which the Client from Assignment 1 can communicate with. Figure 1 shows a high-level overview of the main components you have to implement in Assignment 2.

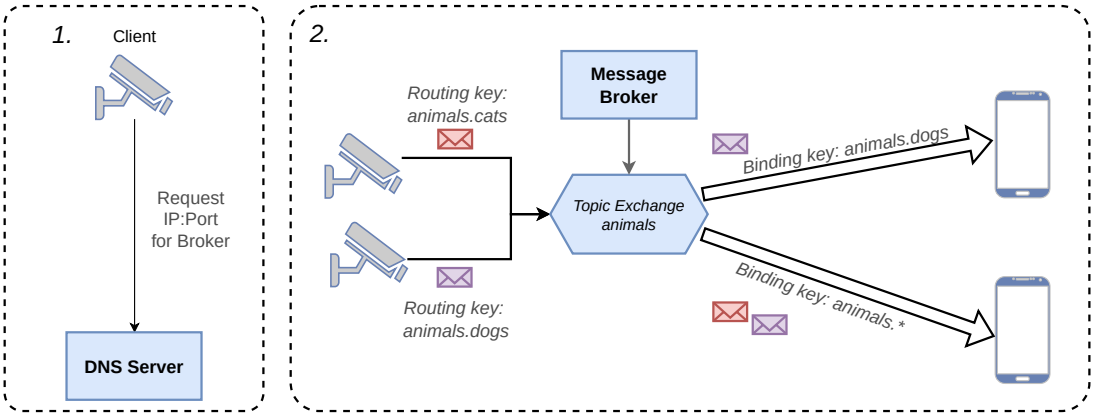

Figure 1: Assignment 2 - Bird's Eye View

2.1 Application Overview

The core components are:

1. The **DNS server** which resolves a domain name (e.g., `my-smqp-broker.com`) into the Message Brokers address, consisting of an IP address and port. *You will have to update your Client implementation to fetch the Message Brokers address.*
2. The **Message Broker** contains the logic for handling exchanges, queues and client connections. Clients can connect to the Message Broker to create or use these exchanges and queues. It also registers its domain with the DNS server on startup. This Message Broker is mostly compatible with the Client you implemented in Assignment 1. The Client must be extended with a few minor extensions. Refer to 2.3.5 for more information. After the extensions your Client should be fully compatible with the Message Broker.

The remaining section explains each of the servers in more detail, starting with the DNS server. Refer to Figure 2 for an overview of the interactions between the components (i.e., Client, DNS server and Message Broker).

2.2 DNS Server

This server implements a basic DNS (Domain Name System), which is a fundamental service in computer networks used to map domain names to IP addresses. The DNS server implements a new protocol called Simple DNS Protocol (SDP). It will provide essential functionalities, including the registration of domain names and resolving queries from clients by returning the corresponding IP address and port for a given domain. This will help you to understand how DNS works at a foundational level, including concepts such as name resolution, request handling, and data storage for domain-to-address mappings. The focus will be on building a lightweight DNS server that can manage domain registrations and respond to lookup requests in real-time.

DNS Server Overview:

- Receives and processes SDP commands from clients over a TCP connection.
- Manages the creation and resolution of domain name mappings.
- Supports multiple simultaneous client connections by utilizing multi-threading.
- Enables message brokers to register themselves with the DNS server.

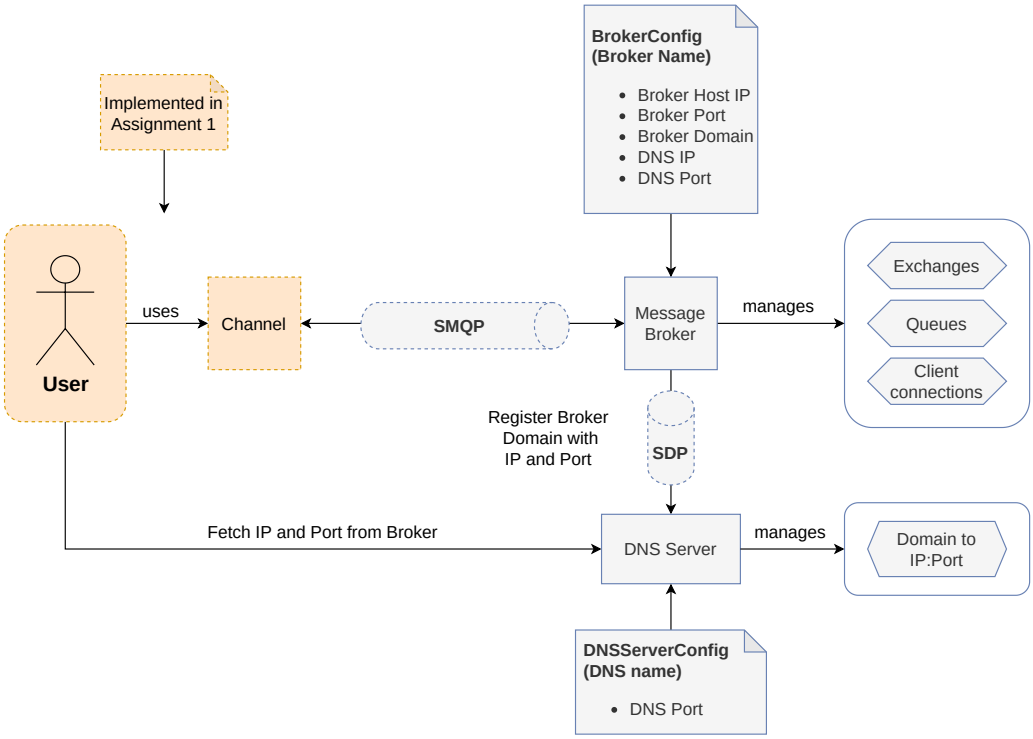

Figure 2: DNS Server and Message Broker Component Interactions

2.2.1 Simple DNS Protocol (SDP)

SDP is a TCP-based plaintext application-layer protocol, that is used by message brokers and clients to deal with domain name resolving. Similar to the SMQP protocol, the client expects upon connection an initial message from the server (i.e., *ok SDP*), as depicted in Figure 3.

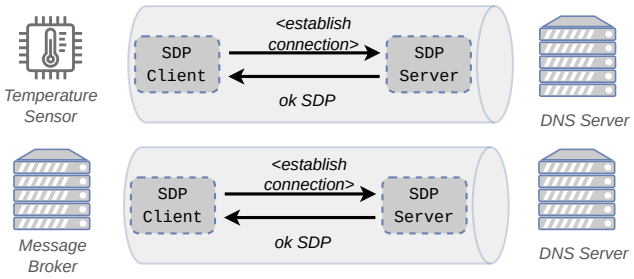

Figure 3: SDP connections.

2.2.2 SDP Key Actions

- **Register Domains**: Associate a domain name with a IP address and port pair.
- **Unregister Domains**: Delete a domain name mapping.
- **Resolve Domains**: Return a IP address and port pair based on a given domain name.

2.2.3 SDP Commands

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

- `register <name> <ip:port>`: Store the domain name to an IP:Port mapping.
  - You can assume that the second argument is of form "*ip:port*".
- `unregister <name>`: Remove the mapping for the given domain name.
- `resolve <name>`: Return the IP:Port mapping for the given domain name.
- `exit`: Disconnects the client from the server.

String Arguments You can assume that arguments do not contain any whitespace, which means you can split the incoming commands by whitespace.

2.2.4 DNS Server Responses

The DNS server has 4 types of responses:

- **Greeting** When a client connects, the DNS server initially sends: `ok SDP` back to the connecting client, indicating that the DNS server is ready and supports SDP.
- **Success** The DNS server responds with the following success commands
  - `register` / `unregister` command: When the DNS server receives a `register` or `unregister` command, the server responds `ok` to indicate successful acceptance and processing of the command.
  - `resolve` command: When the DNS server receives a valid `resolve` command, the server responds with the resolved IP:Port mapping (e.g., `192.168.0.10:8080`).
- **Error** The DNS server responds with the following error commands
  - unknown command: `error usage: <command> <args>` (i.e., when the command is not one of `register`, `unregister` or `resolve`).
  - invalid command: `error <error description>` (i.e., command is correct but contains too many arguments, etc.)
  - unsuccessful processing of command: `error <error description>` (i.e., domain name can't be resolved, conflict at registering domain name, etc.)
- **Disconnect** When a client sends an `exit` command to the DNS server, the server responds with `ok bye` and closes the socket to the client.

2.2.5 Client Session Handling

The following lays out details on how to handle SDP commands during a session:

- Validate `register`, `unregister` & `resolve` arguments by checking if all arguments have been passed and otherwise write an error (i.e., *"error usage: register <name> <ip:port>"*).
- Make sure that your implementation is thread-safe and can deal with multiple clients modifying the same entries concurrently.
- `Unregister` will always return `ok`, therefore also in cases where the domain name entry is not present on the DNS server.

2.2.6 Key Features to Implement

- **Domain Name Registration:** Ability for servers to register new domain names, IP addresses, and ports.
- **Service Unregistration:** Ability for servers to unregister themselves when they go offline.
- **Domain name Resolution:** Allow clients to resolve the IP address and port by the domain name.
- **Concurrency:** Handle multiple simultaneous clients using threads, supporting concurrent registration, unregistration, and resolution requests.
- **In-Memory Persistence:** Maintain an in-memory data structure that holds all registered domain names and their details. The entries are non-persistent, meaning the DNS server boots with zero entries and does not save any when shutting down (i.e., all entries are transient).
- **Error Handling:** Gracefully handle all errors, returning appropriate error messages for invalid commands.
- **Graceful Shutdown:** Ensure that the DNS server can be stopped gracefully, closing all resources properly.

Note for the Submission Interview

While we do not use a distributed name resolution approach in this assignment, familiarize yourself with *recursive* and *iterative* address resolution techniques[a]. In the submission interview, you must be able to explain how they differ in implementation and what their advantages and disadvantages are. You should easily find more literature on this topic, but the Wikipedia article is a good starting point.

[a] https://de.wikipedia.org/wiki/Rekursive_und_iterative_Namensaufl%C3%B6sung

2.3 Message Broker

The Message Broker acts as an intermediary that routes messages between publishers (senders) and subscribers (receivers). The Message Broker will handle requests such as creating exchanges, publishing messages, subscribing to queues, and binding queues to exchanges, as defined by the SMQP. This assignment sets the groundwork for building distributed systems that facilitate communication between multiple clients through message queuing.

You will be responsible for handling the TCP socket communications, managing states of exchanges and queues, and ensuring that messages are correctly routed to the appropriate queues.

Message Broker Overview:

- Automatically registers itself at the DNS server on startup.
- Receive and process SMQP commands from clients over a TCP connection.
- Manage the creation of exchanges (message routing entities), queues (message storage entities), and the relationships (bindings) between them.
- Handle message publication and delivery to subscribers in real-time.
- Supports multiple concurrent connections from publishers and subscribers, realized through multithreading.

2.3.1 Simple Message 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.

2.3.2 SMQP Key Actions

- **Create Exchanges**: Defines a named exchange instance of a specified type. Exchanges are entities that determine where messages are forwarded. They route messages to zero or more queues based on the specified binding key.
- **Create Queues**: Define queues that hold messages for subscribers. Queues hold messages and pass on stored messages once a subscriber has subscribed to the queue.
  - Hold messages when the queue exists, but at the moment no subscriber is subscribed.
- **Bind Queues to Exchanges**: Specify how queues are associated with exchanges using a binding key.
- **Publish Messages**: Accept messages from clients and route them to the appropriate queues by obeying the routing key.
- **Subscribe to Queues**: Allow clients to listen for incoming messages from specific queues.
  - If there are pending messages (i.e., queue is not empty) all messages should be delivered to a subscriber once subscribed.
  - If there is a subscriber, then deliver incoming messages to the subscriber as soon as possible.

2.3.3 SMQP Commands

The client will use a sequence of the following commands **to communicate with the message 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 set queue to the previously set exchange with the specified binding key.
- `subscribe`: Subscribe to the previously specified queue to receive messages. The server-side waits for messages being pushed into the subscribed queue and then writes it to the client socket.
  - Interrupt, if the client sends a `stop` command to the message broker.
  - Interrupt, if the client socket has been closed.
- `publish <routing-key> <message>`: Publish a message to the exchange using the provided routing key. The server-side forwards the incoming message to the appropriate exchange, which pushes it into the bound queues.
- `exit`: Disconnects the client from the message broker.

2.3.4 Client Session Handling

Each client connection is stateful, which means the server keeps a state during the session. This state includes the exchange and queue the client has declared.

The following lays out details on how to handle SMQP commands during a session:

- `Exchanges` are global and shared among client connections. That means if a client wants to declare an exchange, which already exists, it re-uses the exchange instead of a creating a new one. This also applies to `queues` and is fundamental to a properly working message broker.
- Declaring an `exchange` with the same name twice has no effect and re-uses the previously created one. However, this fails if the new one has a different type. In that case, write an error back to the client (e.g., *"error exchange already exists with different type"*).
- Clients can repeat commands during a session. Which simply overrides previous statements. For example, if a client declares a `queue` and then another one (e.g., twice in a row), the first declared queue is overwritten. This does not mean that the queue is deleted but only applies to the clients session state. The same holds for `exchanges`.
- `Queues` deliver messages to subscribed clients. This means that queues actively write into the client connection sockets.
- The `bind` command always uses the most recently declared queue and exchange in the current session. If either is missing, write an error back to the client (e.g., *"error no exchange declared"*, *"error no queue declared"*, etc.).
- `Queues` can be `bound` to multiple exchanges under different binding keys.
- Commands that require an exchange or queue (i.e., `bind`, `subscribe`, `publish`) validate the current session, by checking if all requirements are met. Write an error message back to the client in case the validation fails (e.g., *"error no exchange declared"*, *"error no queue declared"*, etc.). For example, submitting `publish` without prior exchange declaration fails.
- `Publish` uses the previously declared exchange and fails if none has been declared in the current session. However, `publish` does not need a queue.
- `Published` messages with routing keys to which no queue is bound (i.e., "invalid keys") are silently ignored (i.e., the messages are dropped in the background).
- `Subscribe` to the previously defined queue. On successful execution all messages of the corresponding queue will be forwarded to the client subscribed by calling this command. During a `subscription`, the server-side handler of a client does not need to process any commands and stops the `subscription` once a `stop` command has been received from the client.
- `Subscribe` only requires a queue and fails if no queue was declared before.

2.3.5 Assignment 1 Client Extensions

You must adapt your previous client to at some points to make it fully compatible with this assignment. The client is not tested in this assignment, but we recommend to still adapt for easier manual testing of the current assignment.

- Write a `stop` message to the server when stopping the Subscription.
- Fetch the IP address and port of the broker from the DNS server to establish a connection.

2.3.6 Exchange Types

The broker implements four different exchange types. Each type defines a different behaviour of how messages are forwarded to queues. Below is an explanation of each exchange type supported by the message broker. Figure 4 visualizes Direct, Fanout and Topic exchanges.

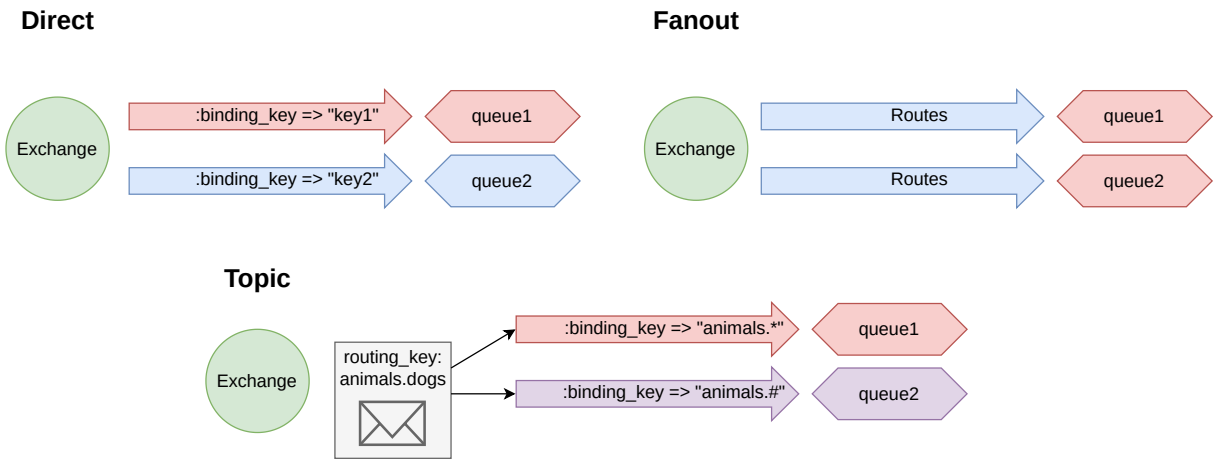

Figure 4: Direct, Fanout and Topic Exchange Types

Direct A direct exchange routes messages to queues based on the routing-key. A direct exchange can be used for unicast and multicast routing of messages.

Details:

- A queue is bound to the exchange with key `k`.
- A message with routing key `r` arrives.
- The exchange forwards the message to all queues where the routing key of the message matches the binding-key of the queue (i.e. where `r=k`).
- If the routing-key of the messages matches no binding key, then the exchange drops the message.

Example:

- Queue `queue1` is bound to the exchange with the binding key `key1`.
- Queue `queue2` is bound to the exchange with the binding key `key2`.
- Message `msg1=(routing-key=key1, msg=some-text-1)` is sent to the exchange.
- `queue1` receives `some-text-1`.
- Message `msg2=(routing-key=key2, msg=some-text-2)` is sent to the exchange.
- `queue2` receives `some-text-2`.
- Message `msg3=(routing-key=key3, msg=some-text-3)` is sent to the exchange.

- No queue receives `some-text-3`.

Note: Messages without a matching routing key will not be delivered to any queue!

Default The default exchange is a special type of direct exchange pre-declared by the broker. Every queue that is created is automatically bound to the default exchange with a binding key which is the same as the queue's name (binding key = queue name). The standard name of the exchange is simply `default`.

Example:

- Queue `queue1` is created and automatically bound to the default exchange, with binding key `queue1`.
- Queue `queue2` is created and automatically bound to the default exchange, with binding key `queue2`.
- Message `msg1=(routing-key=queue1, msg=some-text-1)` is sent to the exchange.
- `queue1` receives `some-text-1`.
- Message `msg2=(routing-key=queue2, msg=some-text-2)` is sent to the exchange.
- `queue2` receives `some-text-2`.
- Message `msg3=(routing-key=key1, msg=some-text-3)` is sent to the exchange.
- No queue receives `some-text-3`.

Note: No additional configuration is required to use the default exchange.

Fanout A fanout exchange routes messages to all queues that are bound to it, regardless of the routing key. If N queues are bound to a fanout exchange, when a new message is published to that exchange a copy of the message is delivered to all N queues. This exchange type is useful when the same message needs to be sent to multiple consumers, such as in broadcasting scenarios.

Example:

- Queue `queue1` and `queue2` are both bound to the fanout exchange.
- Message `msg1=(routing-key=none, msg=some-text-1)` is sent to the exchange. The routing key is ignored.
- `queue1` and `queue2` receive `some-text-1`.

Topic A topic exchange routes messages to queues based on pattern matching between a message routing key routing key and the binding key pattern that was used to bind a queue to an exchange. The routing key in the message can contain multiple words, separated by dots (`.`). Binding keys can contain 2 types special characters:

- `*` (matches exactly one word)
- `#` (matches zero or more words)

Example:

- Queue `queue1` is bound to the exchange with the binding key `animals.*`.
- Queue `queue2` is bound with the binding key `animals.#`.
- Message `msg1=(routing-key=animals.dogs, msg=some-text-1)` is sent to the exchange.
- `queue1` and `queue2` receive `some-text-1`.
- Message `msg3=(routing-key=animals.cats.shorthair, msg=some-text-2)` is sent to the exchange.
- `queue2` receives `some-text-2`.

Topic Exchange Implementation While we do not force you to implement the Topic Exchange in a particular way, we highly recommend to look into using a Trie. The Trie (prefix tree) data structure is used for storing binding key patterns in some real world AMQP topic exchange implementations. Each node in the Trie corresponds to a segment of the binding key, and the Trie allows for efficient storage and retrieval of queues associated with binding key patterns.

Note for the Submission Interview

During the submission interview you must be able to describe why a Trie, or similar data structures such as a deterministic finite automate (DFA), are recommended to implement this feature.

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 can specify any arbitrary routing key (i.e., *none*) as the routing/binding key to keep the examples 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 the TCP sockets, via the plain-text-protocol, by whitespace.

2.3.7 Message Broker Responses

The message broker has 3 types of responses:

- **Greeting:** When a client connects, the message broker initially sends: `ok SMQP`, indicating that the message broker is ready and supports SMQP.
- **Success:** When the server receives a valid command, the server responds with `ok`.
- **Error:** Upon receiving an invalid or unknown command, the server responds with `error <error description>`.

2.3.8 Key Features to Implement

- **Exchange Management:** The server should support different types of exchanges (i.e., direct, default, fanout, topic) and handle their unique routing behaviors. *SMQP* does not support removing exchanges.
- **Queue Management:** The server must manage the creation of queues and bind them to exchanges as per client instructions. *SMQP* does not support removing queues.
- **Message Routing:** Based on the type of exchange, messages published by clients should be routed to the appropriate queues and delivered to subscribers. Messages can be stored temporarily in a queue if no subscriber is available.
- **Subscription Handling:** The server should maintain connections with subscribing clients and deliver messages in real-time.
- **DNS Registration:** Upon startup, the message broker should try to register itself with the DNS server.
- **Default Exchange**: Should be created upon broker startup.
- **Error Handling:** The server should gracefully handle errors such as invalid commands, missing arguments, or disconnected clients.

Note for the Submission Interview

Queue-Design: In the submission interview you must be able to explain the difference between the queue pushing messages to subscribers and subscribers pulling messages from the queue. Also explain what the advantages and disadvantages of each implementation are.

2.4 General Implementation Details

The following implementation details hold for both servers (i.e., DNS Server and Message Broker).

2.4.1 Key Points for Server Socket Handling

- Listening for Connections: The server listens on a specific port, ready to accept client connections. This is managed by the ServerSocket[3][4].
- Handling Multiple clients: To handle multiple clients, the server needs to use threads, where each client is served in its own thread. We recommend Virtual Threads for client-handler-threads for better I/O performance and less expensive thread creation for short-lived threads[5].
- Use an appropriate `ThreadFactory` or `ExecutorService` to create client handler threads. Java provides for both implementations that spawn Virtual Threads[6][7].
- Handling Commands: The server should parse incoming messages (e.g., SMQP commands), process them according to the protocol, and send appropriate responses to clients.

Note for the Submission Interview

In the submission interview you must be able to explain the difference between platform threads and virtual threads in detail, as well as their advantages, disadvantages and common use cases.

2.4.2 Concurrent Client Support

The servers must be able to support multiple concurrent clients. That means, multiple individual clients should be able to issue commands to the server without having to wait for others to finish first. To this end, use Java's `ServerSocket` to accept new clients and handle each in the background. When you receive a new socket connection via `ServerSocket.accept()`, you can use the socket's `InputStream` and `OutputStream` to communicate with the client. As you will be using these streams a lot, it makes sense to encapsulate them into a separate object and provide methods to read and write strings via a common interface.

2.4.3 Concurrent Access & Thread Safety

Many parts of your code will be subject to concurrent access and will require you to think about thread safety. Provide thread safety where necessary, but avoid simply declaring each method as synchronized. Also, using a concurrent collection from the Java standard library (such as `ConcurrentHashMap`), does not automatically solve all concurrency issues.

Note for the Submission Interview

During the submission interview you must be able to explain in detail how concurrent access affects your implementation, and how your implementation guarantees thread safety, while not sabotaging performance unnecessarily.

[3] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/net/ServerSocket.html
[4] https://docs.oracle.com/javase/tutorial//networking/sockets/
[5] https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html
[6] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Thread.html
[7] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/Executors.html#newVirtualThreadPerTaskExecutor()

2.4.4 Server Shutdown

Each server provides a shutdown method. When your server shuts down, you should properly close all resources. Do not simply force your application to end via `System.exit`. Closing the ServerSocket(s) is a good place to start. It does not disconnect any clients, but the server stops accepting new connections. You should then proceed to terminate all open Socket connections. Close any other I/O resources you may be using and shut down all your thread pools. If your application does not terminate after exiting the main method, you may still have some threads running which did not properly terminate.

3 Examples

3.1 Message Broker Examples (SMQP Protocol)

3.1.1 Example 1: Broadcasting Messages with Fanout Exchange

In this example, the message broker uses a **fanout exchange** to broadcast a message to multiple queues.

```
exchange fanout myFanoutExchange                                    1
queue queue1                                                        2
bind none                                                           3
queue queue2                                                        4
bind none                                                           5
publish none Fanout-Broadcast-Message                               6
```

Explanation

- A **fanout exchange** called `myFanoutExchange` is created.
- Two queues, `queue1` and `queue2`, are bound to the exchange. Binding keys are not needed for fanout.
- The message `"Fanout-Broadcast-Message"` is published to all bound queues.

3.1.2 Example 2: Routing Messages with Direct Exchange

This example illustrates how a message is routed to a specific queue using a **direct exchange**.

```
exchange direct myDirectExchange                                    1
queue queue1                                                        2
bind key1                                                           3
queue queue2                                                        4
bind key2                                                           5
publish key1 Direct-Message-for-Key1                                6
```

Explanation

- A **direct exchange** called `myDirectExchange` is created.
- `queue1` is bound with `key1`, and `queue2` with `key2`.
- The message `"Direct-Message-for-Key1"` is routed only to `queue1`, as the routing key matches `key1`.

3.1.3 Example 3: Using Default Exchange

Here, the message broker uses the **default exchange** to route messages to a queue with the same name as the routing key.

```
exchange default default                                            1
queue queue1                                                        2
publish queue1 Default-Exchange-Message                             3
```

Explanation

- `queue1` is automatically bound to the **default exchange**.
- The message `"Default-Exchange-Message"` is routed to `queue1`, as its routing key matches the queue name.

3.1.4 Example 4: Routing Based on Patterns with Topic Exchange

This example demonstrates how a **topic exchange** routes messages based on pattern matching.

```
exchange topic myTopicExchange                                    1
queue queue1                                                      2
bind animals.*                                                    3
queue queue2                                                      4
bind animals.dogs                                                 5
publish animals.dogs Message-for-Dogs                             6
publish animals.cats Message-for-Cats                             7
```

Explanation

- A **topic exchange** called `myTopicExchange` is created.
- `queue1` is bound with the pattern `animals.*`, and `queue2` is bound with `animals.dogs`.
- The message `"Message-for-Dogs"` is routed to both `queue1` and `queue2`, as both match `animals.dogs`.
- The message `"Message-for-Cats"` is only routed to `queue1`, as its binding pattern `animals.*` matches `animals.cats`.

3.2 DNS Server Examples (SDP Protocol)

3.2.1 Example 5: Registering a Domain for the Message Broker

In this example, the message broker registers its domain with the DNS server.

```
register my-broker.com 192.168.1.10:8080                          1
```

Explanation

- The broker registers the domain name `my-broker.com` with the IP `192.168.1.10` and port `8080`.
- This allows clients to resolve the domain to establish communication with the broker.

3.2.2 Example 6: Resolving a Domain Name to Connect to a Broker

This example shows how a client resolves the broker's domain to retrieve its IP address and port.

```
resolve my-broker.com                                             1
```

DNS Server Response

```
192.168.1.10:8080                                                 1
```

Explanation

- The client sends a resolve command for `my-broker.com`.
- The DNS server responds with the IP `192.168.1.10` and port `8080`, allowing the client to connect to the broker.

3.3 Outlook

This assignment builds a message broker that has support for most common message broker features. Your client from Assignment 1 should be able to work with it in an end-to-end manner, albeit with some small changes. The next assignment will introduce you to leader election processes and how UDP transport can play a role in monitoring.

4 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.

4.1 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.

4.1.1 Files and Paths you must leave unmodified

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

If you slightly modify protected files, we may replace your files with the original ones and re-run a Grading Workflow, expect point deduction in this case. In cases of major or severe modifications, point deductions, up to and including 0 points, can be expected. Re-running the Grading Workflow can also be performed, when code base does not properly reflect the Assignment Templates defaults.

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.

4.2 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 100**, which is then **translated to 10 points** (i.e., dividing by 10). The translated assignment points will be uploaded to TUWEL after the assignment deadline has passed.

Re-runs of the Grading Workflow: Re-runs of the *Grading Workflow* for the same commit do not transmit the results to GitHub Classroom. Therefore, your final assignment grade will only consider scores achieved from "regular" Workflow executions (i.e., triggered by commit and push to `main` branch).

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.

4.3 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)

4.4 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.**

5 Submission

Commit and push your code solution to the `main` branch of the repository provided by GitHub Classroom. The score achieved from the last completed "regular" *Grading Workflow* execution, is the one that will count toward the final assignment grade. Refer to Section 4.2 for more details.

5.1 Checklist

- ☐ Ensure that you have not altered any protected resources. Replace them with the original ones when you have accidentally altered them.
- ☐ Ensure that any merge request caused by an update of the Assignment Template is properly reflected in your repository and your final submission.
- ☐ Commit and push your solution ahead of the deadline.
- ☐ Check if the tests are passing as expected in the Grading Workflow.

5.2 Interviews

There is a dedicated submission interview for Assignment 2, that will also talk about the concepts from Assignment 1. The interviews will be conducted in a one on one fashion with a tutor online via Zoom. You must register in time for a interview slot via TUWEL. The registration will be available from Monday, 25.11.2024 @18:00 and close on Friday, 29.11.2024 @18:00.

For any questions, please e-mail us via dslab@dsg.tuwien.ac.at.

Note for the Submission Interview

The submission interview is mandatory and failing to register in time or miss it without a valid excuse (e.g., illness) leads to a full point deduction of Assignment 2.