Exercise 6

MOM (RabbitMQ)

Author: Mariusz Fras

Objectives of the exercise

1. Getting to know the basic architecture of an example solution for MOM - the RabbitMQ broker
2. Getting to know selected typical schemes of application cooperation through MOM.
3. Acquiring the practice of implementing clients of the RabbitMQ broker.

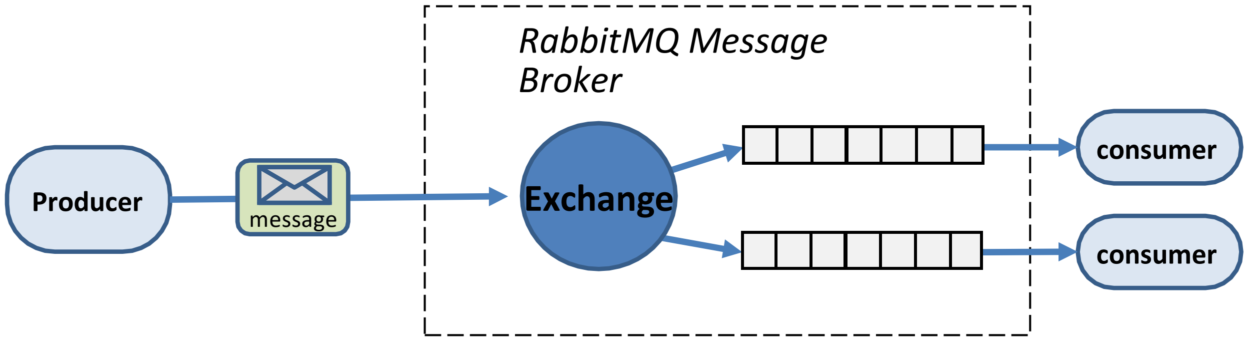
1 Basics of the RabbitMQ boker architecture

RabbitMQ is an open-source message broker for the implementation of MOM - an intermediary layer for distributed applications with communication using message queues.

RabbitMQ supports several message queuing protocols including AMQP (Advanced Message Queuing Protocol), which is widely used and is the basis for the implementation in this exercise.

The basic elements of the architecture for the RabbitMQ broker and the AMQP protocol (see figure) are:

- ***Producer***(or Publisher) – broker client sending messages.
- ***consumers***– broker client receiving (consuming) messages.
- ***queue***– queue – a component that stores messages and forwards them to/from connected clients.
- ***message***– transmitted message consisting of transferred data (payload) and attributes. Some are used by the broker and others (most) for interpretation by applications. Examples of attributes:
  - *Routing key*– used to select messages for routing to queues in various scenarios.
  - *Content-type*
  - *delivery mode*- message persistence.
  - *message priorities*
  - *Expiration period*
- ***Exchange***– an element to which messages are sent and which handles message exchange tasks – distributes them to queues according to specific rules called binding).
- ***binding***– rule that Exchange uses for message distribution. In order for Exchange to route messages to a given queue, it must be bound to it.



Exchange have attributes - e.g.:

- *Name*- name.
- *durability*– durability due to broker stoppages.
- *auto-delete*– deleting Exchange when the last queue is disconnected.

The following types of Exchange are distinguished:

- *direct*– delivers messages to queues according to Routing key.
- *fanout*– delivers messages to all bound queues (Routing key is ignored)

- *Melt*– routes messages to multiple queues according to the Routing key and the template used to bind the queue and Exchange.
- *Headers*– intended for routing according to many attributes (message headers).
- *Default*– default type without definition of the name predeclared by the broker. The important property is: every queue you create is automatically linked to this Exchange with the Routing key value equal to the name of the queue. Result: you can communicate only by specifying the name of the given queue. It is useful for simple applications.

AMQP is a programmable protocol in the sense that AMQP elements and routing schemes are primarily defined by the applications themselves.

2 Runtime and production environment

2.1 RabbitMQ broker installation

The application implementing elements of the queuing system requires the action of the message broker(s). The operation of the application requires the installation/starting of the broker. In this exercise, it is the RabbitMQ broker/server.

- **Installing RabbitMQ**:

- is described here: https://**www.rabbitmq.com/download.html**.

- for Windows the description is here: https://**www.rabbitmq.com/install-**

windows.html.The simplest (proposed) solution is to run the broker in a container **Docker**(finished image from the official repository)

- **Docker installation**:

It is recommended to install the platform in the form of Docker Desktop:

- described here: https://docs.docker.com/get-docker/

- for Windows here: https://docs.docker.com/desktop/install/windows-install/

Another possibility is to install Docker Engine only:

- description is here: https://docs.docker.com/engine/install/binaries/
But this is the option suggested mainly for Linux.

- **Pre-props for Windows**

Docker is a Unix (Linux) platform and requires such a subsystem and/or virtualization for Windows. For Windows 10 and 11 it is recommended to install and use the WSL subsystem - Windows Subsystem for Linux (version 2 - WSL2)

- description is here: https://learn.microsoft.com/en-us/windows/wsl/install

Using WSL instead of Windows Hypervisor virtualization does not interfere with some applications that require Windows Hypervisor to be disabled.

2.2 The proposed scenario for installing the necessary components for Windows 10 and 11

<u>The part that requires system administrator privileges</u>

- Installing the WSL2 subsystem:
  - Open/run Power Shell (or cmd prompt) and issue the command:
    ```
    wsl --install
    ```
    This command enables the appropriate system options and installs the subsystem and the Linux distribution (Ubuntu default or specified).
    You can install without a Linux distribution (lower disk memory usage and faster installation). The following command may work:
    ```
    wsl --install --no-distribution
    ```
    or (for Windows 10 - the documentation is unclear if this syntax will work):
    ```
    wsl --install --d none
    ```
  - You may need to update WSL to WSL2 (not in the latest versions of Windows) with the command:
    ```
    wsl --update
    ```
- Installing Docker Desktop
  - Download and install the Docker Desktop for Windows app/platform from:
    https://docs.docker.com/desktop/install/windows-install/

<u>The part that does not require system administrator privileges</u>

- RabbitMQ image installation for Docker container:
  - Launch Docker Desktop
    The application activates Docker Engine and makes its commands available (e.g. in Power Shell)
  - Open Power Shell and execute the command:
    ```
    docker run -it --name rabbitmq -p 5672:5672 -p 15672:15672
               rabbitmq:3.11-management
    ```
    This command:
    – starts the container with the given name rabbitmq with the broker image, and with the provision of (default) ports for broker administration (5672) and for connections/communication with the broker (15672) - both without encryption of communication,
    – if the broker image is not already in the system, it first downloads it from the official repository
  - For other commands, see the documentation on docker.com.
- Starting the broker
  - In the Docker Desktop application, we can start and stop the container. Note: using stop instead of pause will kill the container. Rebooting in Docker Desktop from the image requires re-configuring the listed ports.
  - You can run the container from Power Shell just like the first time you use it (install).

The containerized broker must be running while the written application is running!

2.3 Application implementation (broker clients)

The exercise will implement the application in Java (it is also possible to use other programming languages.

For Java you can use:

- RabbitMQ library – dependency for maven:

```
<dependencies>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.17.0</version>
</dependencies>
```

- Spring for RabbitMQ -

The description of this option will not be included for now!

3 Exercise

The RabbitMQ library will be used directly in the exercise (without Spring support).

Each implementation will consist of two classes: producer and consumer implementing a given communication scheme.

The RabbitMQ broker is assumed to operate on standard ports.

3.1 Simple messaging

For the simplest communication (with default settings) between the broker's clients (i.e. consumer and producer), the following steps are necessary:

1) For the manufacturer:

a) Establishing a connection and channel for communication:
- creating (and then using) an object of the ConnectionFactory classes,
- setting the broker's address,
- creating a Connection object,
- creating a Channel object.

```
ConnectionFactory factory =newConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

This step is repeated in each client, in each communication scheme.

b) Queue declaration for communication using the queueDeclare method:

```
channel.queueDeclare(QUEUE_NAME,false,false,false,null);
```

This feature has many versions. Here are the following parameters:
- queue name
- queue persistence flag (if true, the broker restart will survive)

- exclusive flag for the connection (when true),
- auto-dequeue flag (if true),
- other configuration arguments (not used here).

Here the queue is not tied to Exchange. Exchange will be used default.

c) Publishing the message (sending to the broker) using the basicPublish method:

```
channel.basicPublish("",QUEUE_NAME,null, msg.getBytes());
```

This feature has several versions. Here are the following parameters:
- Exchange name - here an empty name means using the default one.
- Routing Key - here the name of the queue
  (queues are linked to the default Exchange with a key like the name of the queue),
- *properties*– message attributes (not used here).

String variable msg contains the message to be sent (converted to a string of bytes)

implementation:

- Create a producer class (e.g. Producer) and add a main method to it.
- In the main method, add the mentioned implementation steps by providing the appropriate data (e.g. queue name, messages to be sent) and writing the appropriate information to the console.
- Send approx. 10 messages in a loop approx. every 1 sec.
- Finally, close the channel and connection with the close() method.

2) For the consumer:

a) Establishing a connection and channel for communication - just like in the manufacturer.

b) Queue declaration for communication - as in the producer.

c) Creating an object of the DefaultConsumer type. The handleDelivery method of this class will be called when the consumer receives a message from the queue.

```
DefaultConsumer consumer =newDefaultConsumer(channel) {
    public voidhandleDelivery(String consumerTag,
                              Envelope envelope,
                              AMQP.BasicProperties
                              properties,byte[] body)
                                           throwsIOException {
        String msg =newString(body,"UTF-8");
        [here consume message]
    }
};
```

An object of this class will be specified in connection to the

queue. The parameters of the callback method are:
- individual consumer tag,
- message envelope,

- message attributes (header data),
- message – body – (byte array) – application-specific data.

Note: DeliverCallback with one handle method is also often used instead of this class. Sometimes it is more useful due to the use of lambda expressions.

d) Connecting/registering a consumer to the queue using the basicConsume method:

```
channel.basicConsume(QUEUE_NAME,true, consumer);
```

This feature has many versions. Here are the following parameters:
- queue name
- flag for automatic confirmation of receipt of messages (when false you must confirm explicitly with a command).
- indication of an object with a callback method for receiving messages.

implementation:

- Create a consumer class (e.g. Consumer) and add a main method to it.
- In the main method, add the mentioned implementation steps by providing the appropriate data (e.g. queue name, messages to be sent) and writing the appropriate information to the console.
- Finally, close the channel and connection with the close() method.
  Note: implement closure in response to user action (in) so that the client has time to receive messages.

Testing:

- Start the broker with the standard configuration.
- Start the consumer and then the producer and check the operation.
- Run two instances of the consumer and then the producer and check the operation.

3.2 The consumer as a worker and load balancing

To implement load balancing for messages processed by several consumers, it is enough to make a few modifications to the previous solution.

1) Producer:

The manufacturer's code can be almost identical.
Alternatively, change the name of the communication queue to distinguish it.

implementation:

- Create a producer class (e.g. ProducerOfWork) and copy the previous producer's code.
- Remove thread sleep for 1 sec. (in a loop). This time the client will simulate the given message processing time.

2) Consumer

a) Establishing a connection and communication channel - no changes.

b) After creating the channel - declaration of the number of messages currently delivered to the client using the basicQos method.

```
channel.basicQos(1);
```

Here: one message from the queue is forwarded to the client. Each subsequent one will be delivered after confirming the receipt of the previous one.

c) Queue declaration for communication - no changes.

d) A class with a callback method:
- at the end of the handleDelivery method, add instructions to confirm the consumption of the message:

```
channels.basicAck(envelope.getDeliveryTag(),previousAcks);
```

The method accepts a unique tag and a flag of automatic confirmation of previous messages - here we set it to previousAcks=false.

e) Connecting/registering a consumer to the queue using the basicConsume method:

The second parameter of the basicConsume method is set to false. – we do not automatically confirm the consumption of messages.

f) In the handleDelivery method, before confirming the consumption of the message, we insert the code to simulate some processing time by putting the thread to sleep for a given number of milliseconds.

```
... Thread/Sleep(…) ...
```

Note: The millisecond value must be set differently for different consumer instances.

implementation:

- Create a consumer class (e.g. ConsumerWorker) and copy the previous consumer's code.
- Make the listed code modifications.

Testing:

- Run two consumer instances - set a different message processing time for each.
- Then run the manufacturer and check the operation. Pay attention to the number of messages being processed.

3.3 Publish-Subscribe Communication

For the implementation of sending messages to all subscribers is used **Exchange**fanout type.

1) Producer:

The manufacturer's code is almost identical to the previous one, except for 2 modifications:

a) Instead of a queue declaration, there is an Exchange fanout declaration:

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
```

where EXCHANGE_NAME is some name of your choice.

b) Publishing/sending messages is to Exchange, not to a queue:

```
channel.basicPublish(EXCHANGE_NAME,"",null, message.getBytes());
```

implementation:

- Create a producer class (e.g. ProducerPS) and copy the previous producer's code.
- Make the indicated modifications.

2) Consumer

Similarly, the consumer requires 2 modifications from the first version:

a) Instead of declaring the queue:
- declare Fanout type Exchange,
- get an individual, unique queue name (we don't need to know it),
- bind the queue to Exchange.

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName,EXCHANGE_NAME,"");
```

b) For connection/registration of the consumer to the queue with the basicConsume method, use the retrieved name:

```
channel.basicConsume(queueName,true, consumer);
```

implementation:

- Create a consumer class (e.g. ConsumerPS) and copy the code of the first consumer.
- Make the indicated modifications.

Testing:

- Run two instances of the consumer and then the producer and check the operation.

3.4 Routing of messages to selected recipients / subset of messages

Routing of a subset of messages to selected subscribers is done using a key - the Routing Key attribute.

1) Producer:

The manufacturer's code is similar to the previous one with the following modifications:

a) Instead of an Exchange Fanout declaration, there is an Exchange Direct declaration. channel.exchangeDeclare(*EXCHANGE_NAME*, BuiltinExchangeType.*DIRECT*); where EXCHANGE_NAME is some name of your choice.

b) The publication/sending of the message is with the specification of the routing attribute (Routing
key):

```
channel.basicPublish(EXCHANGE_NAME,"some_string",null,
                               message.getBytes());
```

This attribute/key will determine to which recipients the message will be forwarded.

implementation:

- Create a producer class (e.g. ProducerR) and copy the previous producer's code.
- Make the indicated modifications.
- Publish/send messages with different keys - e.g. here: "info", "alert" and "news".

2) Consumer

The consumer requires similar modifications. Compared to the previous version:

a) Instead of an Exchange Fanout declaration, there is an Exchange Direct declaration.

```
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
```

b) In binding the queue to Exchange, a Routing Key is used to determine which messages will be received:

```
channel.queueBind(queueName,EXCHANGE_NAME,"some_string");
```

You can bind multiple times for different sources/messages.

c) For connection/registration of the consumer to the queue, the basicConsume method remains unchanged.

implementation:

- Create a consumer class (e.g. ConsumerR) and copy the previous consumer's code.
- Make the indicated modifications.
- Implement different bindings for different consumer instances. For example, you can enter keys as call parameters (like: "program.exe. key1 key2").

```
public static voidmain(String[] args)throws... {
  ...
  for(String routingkey: args) {
    channel.queueBind(queueName,EXCHANGE_NAME, routingkey);
}
```

- In the handleDelivery method, in the message, display the value of the key using:

```
Envelope.getRoutingKey()
```

Testing:

- Run two or three consumer instances with different routing key values - e.g.:

  ConsumerR info
  ConsumerR info alert
  ConsumerR news

- Then run the manufacturer and check the operation.

3.5 RPC communication

RabbitMQ also supports the widely used RPC interaction pattern. It should be noted that this mechanism should be audited for error handling, in particular for a long non-response to a request for various reasons. Handling these issues is not covered in this exercise.

The RPC implementation using the RabbitMQ broker includes the following extensions compared to the previous examples:

– two queues are used - one for RPC requests, the other for RPC responses ("callback" queue) from the server (consumer),
– queues for responses ("returns") are individual for the client (producer),
– the client in the request-message must specify the address of the return queue.
– responses to request data should be identifiable and requests and responses are provided with the correlationID attribute - a unique identifier,

In the client-producer side diagram below, we implement an internal RPC client class with a call method that takes a procedure call parameter and returns a result, and uses a CompletableFuture to execute the request in another thread and wait for it to complete. In addition, instead of DefaultConsumer, an object of the DeliverCallback class (functional interface) in the form of a Lambda expression is used for receiving messages.

The implementation scheme is as follows:

1) Producer(RPC client):

In the class, we define an internal class (here RPCClient), and in

it: In the constructor:

a) Establishing a connection and communication channel - standard.

The following sub-items are included in the RPC client method:

```
String call(String msg)throws ...
```

b) Create unique: message id and return queue name:

```
String correlationID = UUID.randomUUID().toString();
String replyQName =channels.queueDeclare().getQueue();
```

c) Creating attributes for the request-message - identifier and name of the return queue transferred to the server:

```
AMQP.BasicProperties props =newAMQP.BasicProperties.Builder()
        .correlationId(correlationID)
        .replyTo(replyQName)
        .build();
```

d) Sending request-

message:`channels.basicPublish("",RPC_QUEUE_NAME,props,msg.getBytes("UTF-8"));` `where RPC_QUEUE_NAME is the name of the request-message queue.`

To receive the response, we create a task-object of the CompletableFuture<…> type. The result can be retrieved by calling its get() method. The result will be available when the task is complete - after calling its complete(…) method.

e) Create an object to receive the result:

```
CompletableFuture<String> response =newcompleteablefuture<>();
```

f) Trigger the receipt of the message - register the handler with the method called when the response message arrives - here in the form of a lambda expression in place of the third parameter of the method
basicConsume(String queue, boolean autoAck, DeliverCallback deliverCallback,
CancelCallback cancelCallback)

```
String tag =channels.basicConsume(replyQName,true,
                                  (consumerTag, returnMsg) -> {
  if(returnMsg.getProperties().getCorrelationId().equals(correlationID))r
    esponse.complete(newString(returnMsg.getBody(),"UTF-8"));
  },consumerTag -> { }
);
```

Here: CancelCallback is "empty".

g) At the end of the call method, receive the result of the call (or wait) cancel the channel and return the result:

```
result = response.get();
channels.basicCancel(tag)
;returnresult;
```

In addition, you can "cleverly" release resources used by an object of the RPCCall class by implementing the AutoCloseable interface, which enforces the close() method in the class:

h) Implement the interface for closing the channel and connection after using the RPCCall class object:

```
classRCCclientimplementsAutoCloseable
  {[...here the previously discussed
  code...]public
  voidclose()throwsexception {
    channels.close();co
    nnection.close();
  }
}
```

In the main method of the producer class in the try…catch… block, we create an object of type RPCCall and call its call method with the appropriate parameter(s).

2) Consumer(RPC server):

The consumer requires a few minor modifications from the basic consumer discussed in the earlier examples:

a) Establishing a connection and communication channel - standard.

b) Queue declaration for communication (for RPC requests) – by default, without automatic acknowledgment of message receipt.

c) Creating an object of the DefaultConsumer type - standard, but inside the method **handleDelivery :**

   – we parse the message into the type of the call parameter (here: string to int)

   – we call the calculation method - implemented in the consumer's class (here in the example fibo(int i))

   – finally, we send back a message with the calculated result and confirm the receipt of the message:

```
channels.basicPublish("", properties.getReplyTo(), properties,
                          response.getBytes("UTF-
8"));channels.basicAck(envelope.getDeliveryTag(),fa
lse);
```

   Here: properties is a parameter of the handleDelivery method of the handler.

   Note the queue name passed in the request from the producer client.

d) Finally, registering to receive customer request messages - standard (use request queue name).

In the example, the consumer class contains a fibo method that calculates (here) the value of the Fibonacci sequence.