Laboratory work №5

In this laboratory I needed to implement messaging patterns to build a “number-processing” service. For this exercise I tried to download RabbitMQ program, but there were many problems. Therefore I used RabbitMQ online service, which accepts and forwards messages and downloaded the client library package: import com.rabbitmq.client.ConnectionFactory;

```
import com.rabbitmq.client.Connection;

import com.rabbitmq.client.Channel;
```

The application performs some processing on the number and shows the results to the user, where I used RPC (remote procedure call) to run some function and wait for result. My RPC system consists of client and server classes.

Named the queue: private String requestQueueName = "rpc_queue";

Created a connection to server:

```
    ConnectionFactory factory = new ConnectionFactory();

    factory.setUri(Constants.uri);

    connection = factory.newConnection();

    channel = connection.createChannel();
```

I created a simple *client* class. It's going to expose a method *call* which sends an RPC request. I used an asynchronous pipeline instead of RPC blocking, results are asynchronously pushed to a next computation stage:

```
         fibonacciRpc = new RPCClient();

        response = fibonacciRpc.call(a + "|" + n);

        System.out.println(" [.] Got ' " + response + " ' ");
```

Callback queue:

A client sends a request message and a server replies with a response message. In order to receive a response we need to send a 'callback' queue address with the request.

```
private String requestQueueName = "rpc_queue";

replyQueueName = channel.queueDeclare().getQueue();

BasicProperties props = new BasicProperties

            .Builder()

            .correlationId(corrId)

            .replyTo(replyQueueName)

            .build();
```

```
channel.basicPublish("", requestQueueName, props, message.getBytes());
```

Message acknowledgment:

What happens if one of the consumers starts a long task and dies with it only partly done? Once RabbitMQ delivers a message to the customer it immediately removes it from memory. In this case, if you kill a worker we will lose the message it was just processing.

An ack(nowledgement) is sent back from the consumer to tell RabbitMQ that a particular message has been received, processed and that RabbitMQ is free to delete it. Message acknowledgments are turned on by default. Soon after the worker dies all unacknowledged messages will be redelivered.

```
consumer = new QueueingConsumer(channel);

channel.basicConsume(replyQueueName, true, consumer);

while (true) {

        QueueingConsumer.Delivery delivery = consumer.nextDelivery();

if (delivery.getProperties().getCorrelationId().equals(corrId)) {

   response = new String(delivery.getBody(),"UTF-8");

   break;

}       }
```

- delivery: Marks a message as persistent (with a value of 2) or transient (any other value).
- replyTo: Commonly used to name a callback queue.
- correlationId: Useful to correlate RPC responses with requests, we set it to a unique value for every request. Later, when we receive a message in the callback queue we'll look at this property, and based on that we'll be able to match a response with a request

We need: import com.rabbitmq.client.AMQP.BasicProperties;

RPC will work like this:

- When the Client starts up, it creates an anonymous exclusive callback queue.
- For an RPC request, the Client sends a message with two properties: replyTo, which is set to the callback queue and correlationId, which is set to a unique value for every request.
- The request is sent to an rpc_queue queue.
- The RPC worker (server) is waiting for requests on that queue. When a request appears, it does the job and sends a message with the result back to the Client, using the queue from the replyTo field.
- The client waits for data on the callback queue. When a message appears, it checks the correlationId property. If it matches the value from the request it returns the response to the application.

```
Exp task: static int ipow(int base, int exp){

int result = 1;

     for (int i=1; i<=exp;i++) {

        result *= base;

     } return result;    }
```

In this case I used tokenizer because I had to send two values: base and exp

```
StringTokenizer tokenizer = new StringTokenizer(message,"|");

            int base = Integer.parseInt(tokenizer.nextToken());

            int exp = Integer.parseInt(tokenizer.nextToken());
```

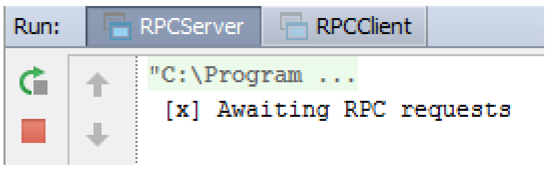
Also I wrote the functions to calculate the factorization, 2 in power of some value (2^x), IsPrime

```
response = "power: " + ipow(base, exp) + "\n" + "binpow: " + binpow(base) + "\n" + "factorial: " +
factorial(base) + "\n" + "isPrime: " + isPrime(base) + "\n";
```

- As usual we start by establishing the connection, channel and declaring the queue.
- We might want to run more than one server process. In order to spread the load equally over multiple servers we need to set the prefetchCount setting in channel.basicQos.
- We use basicConsume to access the queue. Then we enter the while loop in which we wait for request messages, do the work and send the response back.

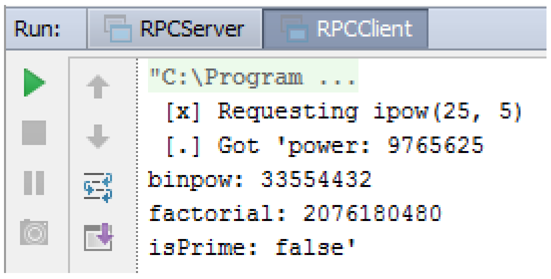
Result:

Server



```
"C:\Program ...
 [x] Awaiting RPC requests
```

Client



```
"C:\Program ...
 [x] Requesting ipow(25, 5)
 [.] Got 'power: 9765625
binpow: 33554432
factorial: 2076180480
isPrime: false'
```