LABWORK
COURSE: DISTRIBUTED SYSTEMS
CHAPTER 4: Communication

1. Develop a RPC system in using RabbitMQ

1.1. Contents

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. RPC uses the client-server model. The requesting program is a client and the service providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned.

RabbitMQ is a messaging broker - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received. In general, RabbitMQ supports message-oriented persistent communication.

In the first part of this labwork, you will construct a RPC system in using RabbitMQ tool.

1.2. Requirements

1.2.1. Theory

- RPC
- Message-oriented communication

1.2.2. Hardwares

- Laptop/PC on Windows

1.2.3. Softwares

- RabbitMQ
- JDK/JRE

1.3. PRACTICAL STEPS

First, you have to install RabbitMQ Server: https://www.rabbitmq.com/install-windows.html

RabbitMQ supports several languages, but in this labwork you'll use Java.

Then, you download three files:

- Client library: http://central.maven.org/maven2/com/rabbitmq/amqp-client/5.5.1/amqp-client-5.5.1.jar

- API Dependencies: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.7.25/slf4j-api-1.7.25.jar
- Simple dependencies: http://central.maven.org/maven2/org/slf4j/slf4j-simple/1.7.25/slf4j-simple-1.7.25.jar
You open the *environment variable* window and do the two things as follows:
- insert a new value to the Path variable: *C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.12\sbin*
(if it's not your case, modify it)
- Add a new system variable named *CP* with the value: *.;amqp-client-5.5.1.jar;slf4j-api-1.7.25.jar;slf4j-simple-1.7.25.jar*

Create a new working folder and place all the three files above into that.
The RPC system we will construct is describes as the figure below:

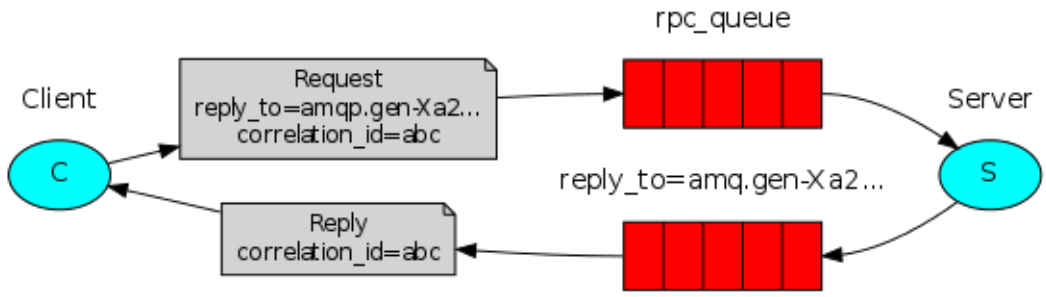

Figure 1: RPC system in using RabbitMQ

Client sends the request to the *rpc_queue* and creates his own queue to wait for the reply of the Server. After receiving a request from the Client, the Server will process the request, and then send the reply to the Client through the queue correspondent. In this labwork, the Server work is to calculate the Fibonacci number.

Now, you are ready to develop Client and Server.

Client:

Go to the working folder where you placed 3 files above.
Create a new class RPCClient in creating a file called RPCClient.java.
In that file, first you have to import all libraries you'll use:

```
import com.rabbitmq.client.AMQP;
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

import java.io.IOException;
import java.util.UUID;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeoutException;
```

After, you declare the class RPCClient with some private attributes:

```
public class RPCClient implements AutoCloseable {

    private Connection connection;
    private Channel channel;
    private String requestQueueName = "rpc_queue";
}
```

You insert now the constructor method for this class:

```
    public RPCClient() throws IOException, TimeoutException {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");

        connection = factory.newConnection();
        channel = connection.createChannel();
    }
```

The *connection* abstracts the socket connection, and takes care of protocol version negotiation and authentication and so on for you. Here you connect to a broker on the local machine - hence the *localhost*. If you wanted to connect to a broker on a different machine you'd simply specify its name or IP address here. Next you create a channel, which is where most of the API for getting things done resides.

You insert now the *call* method. The Client calls the *call* method, and this request will be sent to the Server through the *rpc_queue*.

```
    public String call(String message) throws IOException,
InterruptedException {
        final String corrId = UUID.randomUUID().toString();

        String replyQueueName = channel.queueDeclare().getQueue();
        AMQP.BasicProperties props = new AMQP.BasicProperties
                .Builder()
                .correlationId(corrId)
                .replyTo(replyQueueName)
                .build();

        channel.basicPublish("", requestQueueName, props,
message.getBytes("UTF-8"));

        final BlockingQueue<String> response = new
ArrayBlockingQueue<>(1);

        String ctag = channel.basicConsume(replyQueueName, true,
(consumerTag, delivery) -> {
            if
(delivery.getProperties().getCorrelationId().equals(corrId)) {
                response.offer(new String(delivery.getBody(), "UTF-
8"));
            }
        }, consumerTag -> {
        });

        String result = response.take();
```

```
        channel.basicCancel(ctag);
        return result;
    }
```

In the code above, you will see the *correlation Id* appears. Let's explain it. You see that there is only a single callback queue per client. So that raises a new issue, having received a response in that queue it's not clear to which request the response belongs. That's when the *correlation Id* property is used. You're going to set it to a unique value for every request. Later, when you receive a message in the callback queue you'll look at this property, and based on that you'll be able to match a response with a request. If you see an unknown *correlation Id* value, you may safely discard the message - it doesn't belong to your requests.

Now, you insert the method *close*:

```
    public void close() throws IOException {
        connection.close();
    }
```

You write the code for main method:

```
    public static void main(String[] argv) {
        try (RPCClient fibonacciRpc = new RPCClient()) {
            for (int i = 0; i < 32; i++) {
                String i_str = Integer.toString(i);
                System.out.println(" [x] Requesting fib(" + i_str +
")");
                String response = fibonacciRpc.call(i_str);
                System.out.println(" [.] Got '" + response + "'");
            }
        } catch (IOException | TimeoutException |
InterruptedException e) {
            e.printStackTrace();
        }
    }
```

The Client will send 32 times (from 0-31) to Server to request 32 values of Fibonacci.

Server:

Create the class RPCServer in creating a new file RPCServer.java in your working folder. Put the text below to this file after finishing the code for the method *fib:*

```
import com.rabbitmq.client.*;

public class RPCServer {

    private static final String RPC_QUEUE_NAME = "rpc_queue";

    private static int fib(int n) {
        YOUR-CODE-HERE
    }

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");

        try (Connection connection = factory.newConnection();
             Channel channel = connection.createChannel()) {
```

```
            channel.queueDeclare(RPC_QUEUE_NAME, false, false, false, null);
            channel.queuePurge(RPC_QUEUE_NAME);

            channel.basicQos(1);

            System.out.println(" [x] Awaiting RPC requests");

            Object monitor = new Object();
            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                AMQP.BasicProperties replyProps = new AMQP.BasicProperties
                        .Builder()
                        .correlationId(delivery.getProperties().getCorrelationId())
                        .build();

                String response = "";

                try {
                    String message = new String(delivery.getBody(), "UTF-8");
                    int n = Integer.parseInt(message);

                    System.out.println(" [.] fib(" + message + ")");
                    response += fib(n);
                } catch (RuntimeException e) {
                    System.out.println(" [.] " + e.toString());
                } finally {
                    channel.basicPublish("", delivery.getProperties().getReplyTo(), replyProps,
response.getBytes("UTF-8"));
                    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
                    // RabbitMq consumer worker thread notifies the RPC server owner thread
                    synchronized (monitor) {
                        monitor.notify();
                    }
                }
            };

            channel.basicConsume(RPC_QUEUE_NAME, false, deliverCallback, (consumerTag -> { }));
            // Wait and be prepared to consume the message from RPC client.
            while (true) {
                synchronized (monitor) {
                    try {
                        monitor.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

Question 1: What is the code part that shows that the Server assigns the correlation ID to the response?

Question 2: You base on both code of Client and Server program to explain which code shows that the Client sends request to Server through *rpc_queue* and create a new queue to wait for the reply of the Server.

Compile and Run:

Now it's time to compile and run your program.
First, let's build the Client and Server programs in using the classpath (-cp) assigned to the client library file you have downloaded above:

```
>javac -cp amqp-client-5.5.1.jar RPCClient.java
RPCServer.java
```

Open a command window and run the Server. Make sure that you have add the environment variable *CP*:

```
>java -cp %CP% RPCServer
```

Open a separated command window and run the Client:

```
>java -cp %CP% RPCClient
```

You can try with several Clients, but notice that each one has to run in a separated command window.

Question 3: Now, you try to add the delay to the Server program in inserting this code below the line: `response += fib(n);`

```
try {
    Thread.sleep(2000);
} catch (InterruptedException _ignored) {
    Thread.currentThread().interrupt();
}
```

The Server program will sleep 2 seconds for each request.
Rebuild the Server and run it.
Open several other command window and run the Client on them simultaneously.
Open a separated command window and run the command below:

```
>rabbitmqctl.bat list_queues name messages_ready
messages_unacknowledged
```

What is the result you obtain? Explain it.

2. Effect of QoS on video quality in Video Streaming

2.1. Content

Video streaming over the Internet or over a network has been increasing since many years. This video streaming has been particularly important because of its adoption by several users around the world. The increasing demand to deliver rich multimedia content over the network has also made the video streaming an interesting area for research. There are several areas that have to be taken care for video streaming. These may include delay, bandwidth allocation, packet loss etc. Among these, delay and delay variation (jitter) are important issues as it may result in degradation of quality of video. Due to this the end user may suffer a poor quality of experience.

Video streaming can be done by using several protocols that can be applied to transmit the visual media over the Internet of which UDP, RTP, RTSP, TCP etc are widely used. All these protocols are having their own advantages and disadvantages and among these protocols we are going to use UDP which is at transport layer. UDP is traditionally used protocol for video streaming and

moreover, UDP does not have re-transmission and data-rate management services, which means it is fast enough for real-time audio and video delivery. UDP traffic also enjoys a high-priority status on the Internet, making it fairly smooth and hence we have chosen the same protocol. Moreover there are some other advantages of using UDP like congestion control, rate control, multiplexing etc. The IP protocol is the basic protocol to send the UDP packets over the network. Lot of research has been done on the packet loss and network bandwidth allocation. UDP can also be used for low-bandwidth networks. However there is a lack of research in analyzing the effect of delay and delay variation on video streaming. QoE depends on several factors which include packet loss and transmission delay and we are concentrating on the delay and delay variation (jitter). The online streaming of video requires minimum end-to-end delay/delay variation in order to have a good quality. If the packet does not arrive at the client side on time (due to delay), then it can be treated as a lost packet and so the video frame is also lost, so as the quality of the video is decreased.

In this second part of the labwork, you will setup a testbed with a Video Server and Video Client (both on Linux). From the streaming server we stream the video to the receiver end with the help of a Network Emulator (NetEm) in between. NetEm is a Network Emulator in Linux kernel, which allows dropping, duplicating, delaying of packets. The motivation behind selecting NetEm is that it helps to provide real-time environment.

The lab setup consists of a video streaming server, video client and a Network Emulator "NetEm". The traffic between server and client is forwarded by the network emulator which introduces artificial delay. Experiment is carried out by streaming videos from a streaming server to a client. VLC server and VLC client set up were built on Linux environment. The shaper (NetEm) was installed on the server. The design of the experiment is shown in the Fig. 1.

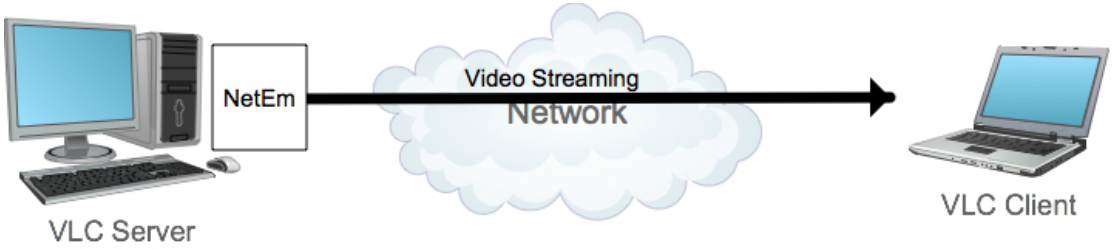

Figure 2: Video streaming in using VLC and NetEm

2.2. Requirements

2.2.1. Theory

- Video Streaming
- QoS

2.2.2. Hardwares

- Laptop/PC on Linux

2.2.3. Softwares

- VLC Client and Server
- NetEm

2.3. PRACTICAL STEPS

Install virtual machine software VirtualBox. Then, install 2 virtual machines on Linux (Ubuntu: http://www.ubuntu.com/download/desktop)
Install VLC software in these 2 virtual machines with the command:

```
$sudo apt-get install vlc
```

First, make sure that your 2 machines can ping each other.

Question 4: What is the IP address of your 2 machines? How to ping each other?

Download some open videos in the site http://www.open-video.org/
Use VLC for video streaming from VLC server to VLC client:
In the Server machine, type the following command:

```
$vlc -vvv input_stream --sout
'#standard{access=http,mux=ogg,dst=SERVER_IP:8080}'
```

where:
input_stream is your video file
SERVER_IP is the IP address of the server

In the Client machine, type the following command:

```
$vlc http://SERVER_IP:8080
```

Question 5: Can you watch the video in the client machine? Evaluate the quality of the video streaming service.

During the video streaming, try to change the network QoS parameters and assess the video quality at the Client PC.

Install the NetEm in the Server machine:

```
$sudo apt-get install iproute
$sudo modprobe sch_netem
```

After, follow these instructions to change the network QoS parameters:

Delay:

You add a fixed amount of delay to all packets going out of the local Ethernet of Server machine:

```
$sudo tc qdisc add dev eth0 root netem delay 100ms
```

Make attention that you have to replace **eth0** with the right network interface of your Server machine. In this labwork, the value **eth0** is used, but you have to replace all of it with your own value.

Now, you do a simple ping test from the server to the client

Question 6: What is the result of the ping test? Can you see an increase of 100 milliseconds?

Question 7: Evaluate the video quality at the Client machine. How can you conclude the impact of fix delay on video streaming service?

From now on, you just change parameters without reloading the *qdisc*.

You know that real wide area networks show variability so it is possible to add random variation:

```
$ sudo tc qdisc change dev eth0 root netem delay 100ms 10ms
```

This causes the added delay to be 100ms ± 10ms. Network delay variation isn't purely random, so to emulate that there is a correlation value as well:

```
$ sudo tc qdisc change dev eth0 root netem delay 100ms 10ms 25%
```

Question 8: Evaluate the video quality at the Client machine. How can you conclude the impact of delay variation on video streaming service?

Packet loss

Random packet loss is specified in the *tc* command in percent. The smallest possible non-zero value is 232 = 0.0000000232%

Now, you let 1 out of 1000 packets to be randomly dropped:

```
$ sudo tc qdisc change dev eth0 root netem loss 0.1%
```

Question 9: Evaluate the video quality at the Client machine. How can you conclude the impact of fix loss rate on video streaming service? Try to increase the value of loss rate to see the impact more clear.

An optional correlation may also be added. This causes the random number generator to be less random and can be used to emulate packet burst losses.

```
$ sudo tc qdisc change dev eth0 root netem loss 0.3% 25%
```

This will cause 0.3% of packets to be lost, and each successive probability depends by a quarter on the last one:
$Prob_n = 0.25 * Prob_{n-1} + 0.75 * Random$

Question 10: Evaluate the video quality at the Client machine. How can you conclude the impact of loss rate variation on video streaming service? Try to increase this value to see the impact more clear.

Packet duplication

Packet duplication is specified the same way as packet loss:

```
$ sudo tc qdisc change dev eth0 root netem duplicate 1%
```

Question 11: Evaluate the video quality at the Client machine. How can you conclude the impact of packet duplication on video streaming service? Try to increase this value to see the impact more clear.

Packet corruption
Random noise can be emulated (in 2.6.16 or later) with the corrupt option. This introduces a single bit error at a random offset in the packet.

```
$ sudo tc qdisc change dev eth0 root netem corrupt 0.1%
```

Packet re-ordering
There are two different ways to specify reordering. The first method gap uses a fixed sequence and reorders every Nth packet. A simple usage of this is:

```
$ sudo tc qdisc change dev eth0 root netem gap 5 delay 10ms
```

This causes every 5th (10th, 15th, ...) packet to go to be sent immediately and every other packet to be delayed by 10ms. This is predictable and useful for base protocol testing like reassembly.
The second form reorder of re-ordering is more like real life. It causes a certain percentage of the packets to get mis-ordered.

```
$ sudo tc qdisc change dev eth0 root netem delay 10ms reorder 25% 50%
```

In this example, 25% of packets (with a correlation of 50%) will get sent immediately, others will be delayed by 10ms.

Question 12: Evaluate the video quality at the Client machine. How can you conclude the impact of packet corruption on video streaming service?

After finishing this labwork, you can remove the affect of NetEm on your network interface in using this command:

```
$ sudo tc qdisc del dev eth0 root
```