BÀI THỰC HÀNH
MÔN HỌC: HỆ PHÂN TÁN
CHƯƠNG 2: Tiến trình và Luồng trong HPT

1. Xây dựng một Chat room sử dụng socket.io

1.1. Nội dung thực hành

Websocket là một giao thức trao đổi thông tin cung ứng cơ chế truyền tin 2 chiều song song trên cùng một kết nối TCP. Giao thức Websocket đã được chuẩn hóa bởi IETF như RFC 6455 năm 2011, và bộ WebSocket API trong Web IDL đã được chuẩn hóa bởi W3C.

Socket.IO là một thư viện JavaScript cho các ứng dụng web thời gian thực. Nó cho phép trao đổi thông tin 2 chiều và thời gian thực giữa các web clients và servers. Nó có 2 phần: thư viện cho phía client chạy trên trình duyệt, và thư viện cho server cho Node.js. Cả 2 thành phần đó đều có chung bộ API. Như Node.js, nó có tính hướng sự kiện. Socket.IO sử dụng giao thức WebSocket, tuy nhiên nó có nhiều chức năng hơn như có khả năng quảng bá tới nhiều sockets, lưu trữ dữ liệu liên quan với mỗi client, và thực hiện trao đổi vào ra bất đồng bộ. Ở bài thực hành này, các bạn sẽ học cách sử dụng Socket.IO (ngôn ngữ Node.js) để phát triển một chat room giao diện web dựa trên giao thức WebSocket. Thực tế đây là một hệ thống thời gian thực với các thao tác gửi và nhận dữ liệu. Bạn không thể thực hiện nó theo các cách truyền thống bởi vì với mô hình web truyền thống, client sẽ yêu cầu server và sau đó server mới trả lời client theo một cách tuần tự. Điều này là không khả thi với một ứng dụng chat khi mà cả 2 bên cần trao đổi với nhau cùng lúc. Với WebSocket, cả client và server có thể gửi dữ liệu cho nhau cùng lúc. WebSocket là một kiểu trao đổi thông tin dạng đường ống mở 2 chiều. Bạn cần phải sử dụng thư viện socket.io để làm điều đó.

1.2. Yêu cầu

1.2.1. Lý thuyết

- WebSocket và thư viện socket.io
- node.js

1.2.2. Phần cứng

- Laptop/PC dùng Windows

1.2.3. Phần mềm

- node.js

1.3. CÁC BƯỚC THỰC HÀNH

Đầu tiên cần phải tải xuống và cài đặt node.js: https://nodejs.org/en/download/

Node.js là một công nghệ back-end sử dụng Javascript được chạy bởi server như PHP, Ruby, hay là Python. Javascript sử dụng các sự kiện. Node.js giữ nguyên những đặc điểm đó vì vậy rất dễ dàng để có thể chạy các đoạn mã bất đồng bộ. Node.js có có bộ quản lý gói riêng là: *npm*. Nó khiến cho việc cài đặt, cập nhật và xóa các gói trở nên dễ dàng. Trong bài thực hành này, các bạn sẽ sử dụng express.js. Nó là một framework dạng web dựa trên node.js.

Bây giờ hãy tạo một thư mục tên là ChatRoomApp, vào đó và khởi tạo môi trường phát triển bằng các lệnh sau:

```
>mkdir ChatRoomApp
>cd ChatRoomApp
>npm init
```

Gặp câu hỏi nào thì hãy ấn enter cho đến hết.

Câu hỏi 1: Tệp nào vừa xuất hiện trong thư mục ChatRoomApp? Nó được sử dụng để làm gì?

Bạn phải cài một số gói cần thiết để phát triển chat room:

```
>npm install --save express
>npm install --save nodemon
>npm install --save ejs
>npm install --save socket.io
```

Giải thích về 4 gói vừa cài:

- *express*: đây là một micro framework ứng dụng dạng web cho node.js
- *nodemon*: gói này có tác dụng phát hiện ra các thay đổi và tự động restart lại server. Sử dụng nó sẽ rất thuận tiện so với các câu lệnh truyền thống.
- *ejs*: ejs là một template engine dùng để đơn giản hóa việc sinh mã HTML.
- *socket.io*: thư viện cho WebSockets

Trong file package.json, bạn hãy thay đổi mã như sau:

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon app"
  }
```

Bây giờ bạn phải phát triển 2 phần: client và server.

Tạo 1 file app.js mới (nó dùng để chạy server và tất cả các gói) và đưa vào đoạn mã sau:

```
const express = require('express')
const app = express()


//set the template engine ejs
app.set('view engine', 'ejs')

//middlewares
app.use(express.static('public'))


//routes
app.get('/', (req, res) => {
     res.send('Hello world')
})

//Listen on port 3000
server = app.listen(3000)
```

Bây giờ hãy chạy ứng dụng của bạn với lệnh:

```
>npm run start
```

Câu hỏi 2: Mở trình duyệt và gõ vào đó địa chỉ *http://localhost:3000,* bạn sẽ nhận được thông điệp gì?

Thêm các đoạn mã sau vào file *app.js*:

```
//socket.io instantiation
const io = require("socket.io")(server)


//listen on every connection
io.on('connection', (socket) => {
         console.log('New user connected')
})
```

Ở đây, đối tượng *io* sẽ cho phép chúng ta sử dụng thư viện *socket.io*. Đối tượng *io* bây giờ sẽ nghe trên mỗi kết nối đến với ứng dụng của bạn. Mỗi lần có một người dùng mới kết nối đến, nó sẽ hiển thị thông điệp "New user connected" lên console.

Câu hỏi 3: Bạn hãy thử reload (Ctrl-R) lại trình duyệt. Bạn có nhìn thấy gì mới xuất hiện trên cửa sổ không? Nếu không có gì xuất hiện hết thì là vì sao?

Bây giờ thì socket.io mới được cài đặt trên phần server. Kế tiếp, chúng ta sẽ làm điều tương tự trên phần client.

Bạn chỉ cần thay đổi một dòng trong file *app.js*. Cụ thể, bạn không muốn hiển thị thông điệp "Hello world" nữa, nhưng một cửa sổ thật với các ô chat box, ô nhập

vào username, thông điệp, và nút Send. Để làm được điều đó bạn phải có một file html (trong trường hợp này là file *ejs*) khi truy cập vào thư mục root "/".

Thay đổi thành phần routes trong file *app.js* như sau:

```
//routes
app.get('/', (req, res) => {
        res.render('index')
})
```

Trong thư mục ChatRoomApp của bạn, hãy tạo 2 thư mục con mới tên là *public* và *views*.

```
>mkdir public
>mkdir views
```

Trong thư mục views, hãy tạo tệp tên là index.ejs
Trong thư mục views, hãy tải file index.ejs từ đường link:
https://github.com/anhth318/ChatRoomApp/blob/master/views/index.ejs,
hoặc tự tạo file và viết nội dung sau:

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" const="text/html;charset=UTF-8"
/>
    <link href="http://fonts.googleapis.com/css?family=Comfortaa"
rel="stylesheet" type="text/css">
    <link rel="stylesheet" type="text/css" href="style.css" >
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.0.4/socket.io
.js"></script>
    <title>Distributed Systems Course</title>
  </head>

  <body>
    <header>
      <h1>Chat room for Distributed Systems class</h1>
    </header>

    <section>
      <div id="change_username">
                    <input id="username" type="text" />
                    <button id="send_username" type="button">Change
username</button>
      </div>
    </section>

    <section id="chatroom">
      <section id="feedback"></section>
    </section>



    <section id="input_zone">
      <input id="message" class="vertical-align" type="text" />
```

```
      <button id="send_message" class="vertical-align"
type="button">Send</button>
    </section>

    <script src="http://code.jquery.com/jquery-
latest.min.js"></script>
    <script src="chat.js"></script>
  </body>
</html>
```

Trong thư mục *public*, hãy tạo file *chat.js* với nội dung sau:

```
$(function(){
     //make connection
     var socket = io.connect('http://localhost:3000')
});
```

(chú ý là nếu bạn làm việc với 2 máy tính thì hãy thay localhost bằng địa chỉ IP của server).

Tải file *style.css* và đặt nó trong thư mục *public*:
https://github.com/anhth318/ChatRoomApp/blob/master/public/style.css

Câu hỏi 4: Refresh trang *localhost:3000*, bạn nhìn thấy thông điệp nào?

Ở phía client, bây giờ bạn muốn làm chủ một số hành động khác như: gửi tin nhắn, nghe tin nhắn mới, và gửi đi username. Thay đổi mã nguồn của *chat.js* như sau:

```
$(function(){
   //make connection
   var socket = io.connect('http://localhost:3000')

   //buttons and inputs
   var message = $("#message")
   var username = $("#username")
   var send_message = $("#send_message")
   var send_username = $("#send_username")
   var chatroom = $("#chatroom")

   //Emit message
   send_message.click(function(){
       socket.emit('new_message', {message : message.val()})
   })

   //Listen on new_message
   socket.on("new_message", (data) => {
       message.val('');
       chatroom.append("<p class='message'>" + data.username + ": " +
data.message + "</p>")
   })

   //Emit a username
   send_username.click(function(){
          socket.emit('change_username', {username : username.val()})
   })
});
```

Trong file app.js, hãy thay đổi mã nguồn của phần *listen on every connection* như sau:

```
//listen on every connection
io.on('connection', (socket) => {
    console.log('New user connected')

    //default username
    socket.username = "Anonymous"

    //listen on change_username
    socket.on('change_username', (data) => {
          socket.username = data.username
    })

    //listen on new_message
    socket.on('new_message', (data) => {
        //broadcast the new message
        io.sockets.emit('new_message', {message : data.message,
username : socket.username});
    })
})
```

Với sự kiện *new_message,* bạn có thể thấy chúng ta gọi đặc tính socket của đối tượng *io*. Nó biểu diễn tất cả các socket đã kết nối. Vậy dòng này sẽ gửi một tin nhắn đến tất cả các sockets. Chúng ta muốn hiển thị tin nhắn được gửi bởi một user lên màn hình của tất cả các users khác (và của chính user gửi

Bây giờ hãy test ứng dụng của bạn. Mở đường dẫn http://localhost:3000/ trong một vài tab của trình duyệt của bạn, thử chat trong room.

Nếu bạn muốn hiển thị dòng "User A is typing ..." khi đang có người dùng khác gõ thì bạn có thể làm như sau: Mở file *app.js* , thêm vào đoạn mã sau dưới hàm *io.on()*

```
//listen on typing
socket.on('typing', (data) => {
         socket.broadcast.emit('typing', {username :
socket.username})
})
```

Mở file *chat.js*, thêm vào mã như sau:

Trong phần "//buttons and inputs", thêm vào dòng sau:

```
var feedback = $("#feedback")
```

Trong phần "//Listen on new_message", sửa lại code như sau:

```
//Listen on new_message
socket.on("new_message", (data) => {
    feedback.html('');
    message.val('');
    chatroom.append("<p class='message'>" + data.username + ": " +
data.message + "</p>")
})
```

Thêm vào 2 phần như sau:

```
//Emit typing
message.bind("keypress", () => {
    socket.emit('typing')
})

//Listen on typing
    socket.on('typing', (data) => {
    feedback.html("<p><i>" + data.username + " is typing a
message..." + "</i></p>")
})
```

Bây giờ bạn có thể thử refresh lại cửa sổ trình duyệt của bạn để thử nghiệm chat room.

Câu hỏi 5: Bây giờ bạn hãy thử gõ gì đó lên một tab. Cùng lúc đó, nhìn sang tab khác của người dùng khác, bạn thấy gì?

2. Phát triển hệ thống RPC sử dụng RabbitMQ

2.1. Nội dung

Remote Procedure Call (RPC) là một giao thức mà một chương trình có thể sử dụng để yêu cầu dịch vụ từ một chương trình khác nằm trên một máy tính khác. RPC sử dụng mô hình client-server. Chương trình yêu cầu được gọi là client và chương trình cung ứng dịch vụ được gọi là server. Được ví như một lời gọi thủ tục cục bộ, RPC đồng bộ sẽ làm dừng chương trình tới bao giờ nhận được câu trả lời từ server.

RabbitMQ là một công cụ broker dành cho thông điệp (công cụ xử lý trung gian cho các thông điệp). Nó cung ứng cho các ứng dụng của bạn một platform chung để gửi nhận các thông điệp, và đảm bảo các thông điệp đó được lưu trữ an toàn đến khi đến được với bên nhận. Nhìn chung, RabbitMQ hỗ trợ cơ chế trao đổi thông tin hướng thông điệp bền vững, đây chính là nội dung lý thuyết chúng ta đã được học trên lớp.
Ở phần đầu của bài thực hành này, các bạn sẽ xây dựng một cơ chế RPC bằng cách sử dụng công cụ RabbitMQ.

2.2. Yêu cầu

2.2.1. Lý thuyết

- RPC
- Message-oriented communication

2.2.2. Phần cứng

- Laptop/PC on Windows

2.2.3. Phần mềm

- RabbitMQ
- JDK/JRE

2.3. Các bước thực hành

Đầu tiên bạn phải cài đặt RabbitMQ Server: https://www.rabbitmq.com/install-windows.html
RabbitMQ hỗ trợ nhiều ngôn ngữ, nhưng trong bài thực hành này chúng ta sẽ sử dụng Java.

Bây giờ bạn sẽ tải 3 files sau:
- 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
Hãy mở cửa sổ quản lý biến môi trường (*environment variable*) và làm 2 điều sau:
- Hãy thêm vào giá trị cho biến Path: *C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.12\sbin*
(Nếu giá trị của máy bạn khác thì sửa lại cho đúng)
- Thêm vào 1 biến tên là CP với giá trị sau: *.;amqp-client-5.5.1.jar;slf4j-api-1.7.25.jar;slf4j-simple-1.7.25.jar*
(Bước này chỉ là tùy ý để các bước sau bạn gõ lệnh nhanh hơn thôi).

Hãy tạo 1 thư mục làm việc mới, copy hết 3 files trên vào đó.
Hệ thống RPC mà chúng ta sắp xây dựng sẽ được mô tả như sau:

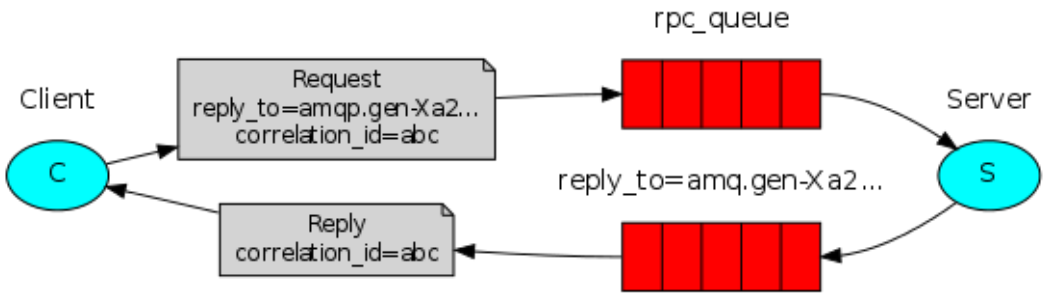

Figure 1: RPC system in using RabbitMQ

Client gửi yêu cầu đến hàng đợi *rpc_queue* và tạo ra một hàng đợi riêng của nó để chờ kết quả trả về từ Server. Sau khi nhận yêu cầu từ Client, Server sẽ xử lý yêu

cầu và sau đó gửi trả về câu trả lời cho Client vào hàng đợi tương ứng. Ở bài thực hành này, Server sẽ vận hành để tính chuỗi Fibonacci.

Bây giờ bạn đã có thể sẵn sàng để xây dựng Client và Server.

Client:

Vào thư mục làm việc mà đang có 3 files vừa rồi. Tạo một lớp RPCClient bằng cách tạo file RPCClient.java. Trong file đó, đầu tiên cần phải import một số thư việc mà bạn sẽ dùng:

```
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;
```

Sau đó, bạn khai báo lớp RPCClient với một vài đặc tính sau:

```
public class RPCClient implements AutoCloseable {

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

Thêm phương thức khởi tạo như sau:

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

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

Thực thể *connection* đại diện cho kết nối socket, và nó sẽ lo việc xác thực và thỏa thuận phiên bản cho bạn. Bạn kết nối tới một broker trong máy cục bộ (localhost). Nếu bạn muốn kết nối đến một broker nằm ở máy khác thì chỉ việc thay localhost ở trên bằng địa chỉ IP của máy đó. Tiếp tới bạn sẽ tạo một kênh (channel), nó là nơi mà hầu hết bộ thư viện API nằm ở đó.

Bây giờ hãy thêm vào phương thức *call*. Client sẽ gọi phương thức *call*, và yêu cầu sẽ được gửi lên cho Server thông qua *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;
    }
```

Ở đoạn code trên bạn sẽ thấy có sự xuất hiện của *corralationID*. Biến này sẽ được sử dụng để giải quyết vấn đề khi mà có duy nhất một hàng đợi cho mỗi Client để nhận câu trả lời từ Server. Và sẽ thật khó phân biện là câu trả lời đó là cho yêu cầu nào, chính vì vậy người ta sử dụng *corralationID* để phân biện câu trả lời là dành cho yêu cầu nào. Trong trường hợp nếu *corralationID* là một giá trị lạ, thì client có thể loại bỏ câu trả lời đó, vì nó không dành cho các yêu cầu mà Client gửi lên.

Giờ hãy thêm phương thức *close*:

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

Bây giờ hãy viết code cho phương thức *main*:

```
    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();
        }
    }
```

Client sẽ gửi 32 lần lên Server để yêu cầu 32 giá trị của chuỗi số Fibonacci.

Server:

Tạo lớp RPCServer bằng cách tạo file RPCServer.java ở trong thư mục làm việc. Thêm đoạn code sau và tự mình hoàn thiện phương thức fib (để sinh ra số thứ n của chuỗi fibbonaci).

```
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();
                    }
                }
            }
        }
    }
}
```

Câu hỏi 6: Đâu là đoạn code mà Server gán *correlationID* vào câu trả lời?

Câu hỏi 7: Dựa vào cả code của Client và Server để giải thích đâu là đoạn code mà Client gửi yêu cầu lên cho Server thông qua hàng đợi rpc_queue và tạo ra một hàng đợi mới để chờ câu trả lời của Server.

Dịch và chạy chương trình:

Bây giờ chúng ta sẽ dịch và chạy chương trình.
Đầu tiên, hãy xây dựng chương trình Client và Server bằng cách sử dụng classpath (-cp) gắn với file thư viện client mà bạn vừa tải về lúc ban đầu:

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

Mở thêm một cửa sổ command và chạy Server. Hãy chắc rằng bạn đã thêm được biến môi trường *CP* như ở trên có hướng dẫn:

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

(ghi chú: trong trường hợp không muốn tạo biến môi trường *CP* như trên thì có thể thay thế *CP* bằng đoạn giá trị đó).

Mở một cửa sổ tách biệt và chạy Client:

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

Bạn có thể thử với nhiều Client, nhưng chú ý là mỗi Client phải chạy trên một cửa sổ riêng biệt.

Câu hỏi 8: Bây giờ hãy thử thêm một chút delay vào chương trình Server bằng cách thêm vào đoạn code sau ở dưới dòng: `response += fib(n);`

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

Chương trình Server sẽ ngủ 2s đối với mỗi request. Hãy dịch lại chương trình Server và chạy nó.
Mở cùng lúc nhiều cửa sổ command và chạy nhiều chương trình Client trên đó cùng lúc.
Cùng lúc đó mở một cửa sổ command khác và chạy dòng lệnh sau:

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

Bạn nhận được kết quả hiển thị gì? Giải thích!

3. Phân tích ảnh hưởng của các thông số QoS lên dịch vụ truyền dòng video

3.1. Nội dung

Truyền phát video qua Internet hoặc qua mạng đã rất phát triển từ nhiều năm trở lại đây. Dịch vụ truyền phát video này đặc biệt quan trọng vì được đa số người dùng trên khắp thế giới sử dụng mỗi ngày. Nhu cầu ngày càng tăng để cung cấp nội dung đa phương tiện phong phú qua mạng cũng khiến video phát trực tuyến trở thành một lĩnh vực thú vị để nghiên cứu. Có một số vấn đề phải được xem xét để đảm bảo chất lượng truyền phát video như độ trễ, phân bổ băng thông, mất gói, v.v. Trong số này, biến đổi độ trễ (jitter) và độ trễ (delay) là những vấn đề quan trọng vì nó có thể gây ra hiện tượng suy giảm chất lượng video. Do đó, người dùng cuối có thể phải chịu chất lượng trải nghiệm kém.
Truyền phát video có thể được thực hiện bằng cách sử dụng một số giao thức có thể được áp dụng để truyền phương tiện trực quan qua Internet trong đó UDP, RTP, RTSP, TCP, v.v được sử dụng rộng rãi. Tất cả các giao thức này đều có những ưu điểm và nhược điểm riêng và trong số các giao thức này, chúng ta sẽ sử dụng UDP ở lớp vận chuyển. UDP là giao thức được sử dụng theo truyền thống để truyền phát video và hơn nữa, UDP không có dịch vụ quản lý tốc độ và truyền lại dữ liệu, điều đó có nghĩa là nó đủ nhanh để phân phối âm thanh và video theo thời gian thực. Lưu lượng truy cập UDP cũng có trạng thái ưu tiên cao trên Internet, khiến nó khá trơn tru và do đó chúng tôi đã chọn cùng một giao thức. Ngoài ra, có một số lợi thế khác của việc sử dụng UDP như điều khiển tắc nghẽn, điều khiển tốc độ, ghép kênh, vv Giao thức IP là giao thức cơ bản để gửi các gói UDP qua mạng. Rất nhiều nghiên cứu đã được thực hiện về việc mất gói và phân bổ băng thông mạng. UDP cũng có thể được sử dụng cho các mạng băng thông thấp. Tuy nhiên, thiếu nghiên cứu trong việc phân tích ảnh hưởng của biến đổi độ trễ và độ trễ đối với truyền phát video. QoE phụ thuộc vào một số yếu tố bao gồm mất gói và độ trễ truyền và chúng tôi đang tập trung vào biến đổi trễ (jitter) và trễ (delay). Việc truyền phát video trực tuyến yêu cầu biến đổi trễ / trễ tối thiểu từ đầu đến cuối để có chất lượng tốt. Nếu gói không đến được máy khách đúng thời điểm (do chậm trễ), thì nó có thể được coi là gói bị mất và do đó khung video cũng bị mất, do đó chất lượng của video bị giảm.

Trong phần thứ hai của bài tập thực hành này, bạn sẽ thiết lập một thử nghiệm với Máy chủ video và Máy khách video (cả 2 chạy trên Linux). Từ Server phát trực tuyến, chúng ta truyền phát video đến đầu thu với việc sử dụng công cụ NetEm ở trung gian. NetEm là Trình mô phỏng mạng trong nhân Linux, cho phép drop, sao chép, trì hoãn các gói tin. Lý do để chúng ta sử dụng công cụ NetEm là nó giúp mô phỏng môi trường mạng thời gian thực.

Testbed cho bài thực hành bao gồm Máy chủ phát video, Máy client, và công cụ NetEm. Dòng video giữa client và server sẽ được truyền qua NetEm và sẽ sinh ra độ trễ nhân tạo. Chúng ta sẽ sử dụng chương trình VLC server và VLC client. Chương trình NetEm được cài trên Server. Mô hình thực nghiệm được mô tả như hình dưới đây.

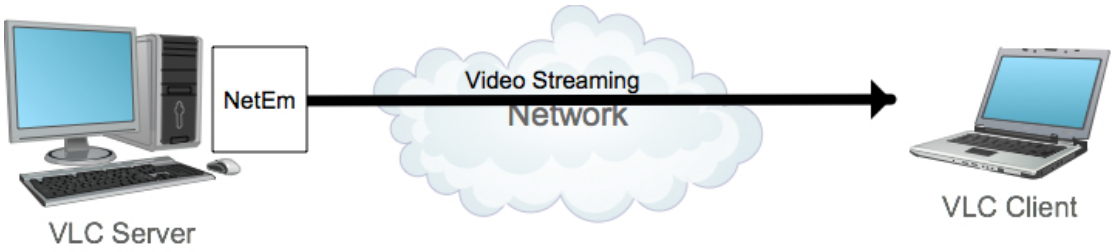

Figure 2: Video streaming in using VLC and NetEm

3.2. Yêu cầu

3.2.1. Lý thuyết

- Video Streaming
- QoS

3.2.2. Phần cứng

- Laptop/PC on Linux

3.2.3. Phần mềm

- VLC Client and Server
- NetEm

3.3. CÁC BƯỚC THỰC HÀNH

Hai máy sử dụng trong bài thực hành có thể là 2 máy ảo (tạo bởi virtualbox) hoặc là 2 máy thật của 2 bạn trong nhóm.
Cài phần mềm VLC lên 2 máy bằng câu lệnh sau:

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

Đầu tiên phải đảm bảo 2 máy có thể kết nối với nhau bằng cách dùng công cụ *ping* để thử.

Câu hoi 9: Địa chỉ IP của 2 máy là gì? Làm sao để *ping* nhau?

Tải một số video về máy Server từ website http://www.open-video.org/
Bây giờ chúng ta sẽ sử dụng VLC để truyền dòng video từ server đến client. Để làm được điều đó thì ở máy server chúng ta cho chạy dòng lệnh sau:

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

trong đó:
input_stream là tên file video của bạn muốn phát
SERVER_IP là địa chỉ IP của Server.

Ở máy Client thì chạy lệnh sau:

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

Câu hỏi 10: Bạn đã xem được video trên máy client chưa? Đánh giá chất lượng video mà bạn xem trên máy client.

Bây giờ trong quá trình truyền dòng video, chúng ta sẽ chủ động thay đổi các thông số QoS để thay đổi chất lượng mạng, từ đó quan sát chất lượng video ở máy client. Để thực hiện điều đó thì chúng ta sẽ sử dụng công cụ NetEm.

Cài đặt NetEm trên máy Server như sau:

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

Bây giờ chúng ta sẽ tiến hành các câu lệnh để thay đổi các thông số QoS. Các câu lệnh tiếp theo đây sẽ thực hiện ở máy Server và có thể gõ song song với quá trình đang truyền video.

Độ trễ (Delay):

Bạn sử dụng câu lệnh sau để thêm vào một chút delay cho tất cả các gói tin đi qua giao diện Ethernet của máy Server:

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

Thay thế eth0 bằng giao diện mạng mà Server thực sự đang sử dụng để kết nối với Client.
Trong bài thực hành này chúng ta coi mặc định eth0 là giao diện mạng sử dụng. Bạn phải thay thế hết chúng cho phù hợp.
Hãy thử thực hiện một lệnh ping từ máy Server đến Client.

Câu hỏi 11: Kết quả nhận được sau lệnh ping là gì? Bạn có thấy độ trễ đã tăng 100ms không?

Câu hỏi 12: Hãy tắt chức năng sử dụng bộ đệm ở máy Client. Sau đó hãy đánh giá chất lượng của video nhận được ở máy Client. Bạn kết luận thế nào về ảnh hưởng của delay với dịch vụ truyền dòng video?

Các lệnh tiếp tới đây chúng ta sẽ chỉ thay thế các giá trị chứ không khởi động lại chương trình *qdisc* (nghĩa là chỉ dùng option *change* thay vì *add*).

Chúng ta biết rằng ở môi trường mạng thật thì các thông số mạng không cố định như vậy, vì vậy chúng ta sẽ thêm vào một giá trị biến đổi cho chúng:

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

Tác dụng của lệnh trên là tạo ra độ trễ thay đổi 100ms ± 10ms. Sự biến đổi giá trị delay của mạng không hoàn toàn là ngẫu nhiên, vì vậy chúng ta sẽ thêm vào một giá trị tương quan:

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

Câu hỏi 13: Cũng như câu hỏi 7, hãy quan sát video ở Client và đưa ra đánh giá và kết luận về ảnh hưởng của độ biến đổi delay lên chất lượng dịch vụ truyền video.

Mất gói tin (Packet loss):

Giá trị mất mát gói tin ngẫu nhiên được xác định trong lệnh tc dựa trên đơn vị phần trăm (%). Giá trị dương nhỏ nhất là 0.0000000232%

Bây giờ hãy thử cho 1/1000 gói tin bị mất ngẫu nhiên:

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

Câu hỏi 14: Hãy xem video ở client và đánh giá về độ ảnh hưởng của packet loss lên chất lượng dịch vụ truyền video. Thử tăng giá trị của tỷ lệ mất gói tin lên để thấy độ ảnh hưởng rõ nét hơn.

Giá trị tương quan có thể đưa vào để làm giảm tính ngẫu nhiên của việc mất gói tin:

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

Câu lệnh trên có nghĩa là 0.3% của các gói tin sẽ bị mất, và mỗi xắc suất để gói tin kế tiếp sẽ phụ thuộc 25% vào xắc suất mất của gói phía trước:
$Prob_n = 0.25 * Prob_{n-1} + 0.75 * Random$

Câu hỏi 15: Hãy xem video ở client và đánh giá về độ ảnh hưởng của việc biến đổi packet loss lên chất lượng dịch vụ truyền video. Thử tăng giá trị của tỷ lệ mất gói tin lên để thấy độ ảnh hưởng rõ nét hơn.

Lặp gói tin (Packet duplication)

Cơ chế lặp cũng tương tự cơ chế mất gói tin:

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

Câu hỏi 16: Hãy xem video ở client và đánh giá về độ ảnh hưởng của việc lặp gói tin lên chất lượng dịch vụ truyền video. Thử tăng giá trị của tỷ lệ lặp gói tin lên để thấy độ ảnh hưởng rõ nét hơn.

Lỗi gói tin (Packet corruption)

Các lỗi ngẫu nhiên có thể được làm giả lập như sau:

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

Lỗi đảo thứ tự gói tin (Packet re-ordering)

Có 2 cách để thực hiện đảo thứ tự các gói tin. Cách đầu tiên là sử dụng chuỗi cố định và đảo gói tin thứ N. Ví dụ như câu lệnh sau:

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

Có nghĩa cứ gói tin thứ 5 (thứ 10, 15, ...) sẽ được gửi ngay lập tức còn các gói tin khác thì bị delay 10ms.
Cách thứ 2 là thay đổi thứ tự một cách ngẫu nhiên, giống môi trường mạng thật hơn. Nó gây ra một số phần trăm nhất định của các gói tin bị đảo lộn thứ tự:

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

Ở câu lệnh trên thì 25% các gói tin (với độ tương quan là 50%) sẽ được gửi ngay lập tức, còn các gói tin khác thì bị chậm 10ms.

Câu hỏi 17: Hãy xem video ở client và đánh giá về độ ảnh hưởng của việc đảo thứ tự gói tin lên chất lượng dịch vụ truyền video.

Sau khi kết thúc bài thực hành, bạn hãy xóa bỏ hiệu ứng của NetEm lên giao diện mạng của bạn bằng lệnh sau:

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