NOVEMBER 22, 2020

2ª Aula Prática
GOOGLE RPC

GRUPO 12
RÚBEN SANTOS - 47766
ALEXANDRE SANTOS - 47926
ANA PEREIRA - 47900

Exercício

A segunda aula prática consiste no desenvolvimento de aplicações Cliente/Servidor em *Google RPC* e na execução da aplicação servidora numa máquina virtual na *Google Cloud Platform* que permite controlar a entrada e saída de veículos (aplicação cliente) com a interface/contrato gRPC.

O problema proposto tem como base uma estrada que contém 5 pontos de acesso, em que cada ponto de acesso é a entrada/saída de veículos, existindo a possibilidade do veículo entrar e sair no mesmo ponto. É feito ainda o pagamento de portagens na estrada consoante o percurso efetuado pelo veículo, o valor desse pagamento é obtido através de uma comunicação a um servidor externo (fornecido pelo professor num ambiente cloud simulando uma entidade bancária) via gRpc.

Considera-se ainda que durante o percurso os veículos podem emitir ou receber avisos de objetos, animais ou qualquer outra situação anormal presente na estrada. Quando o servidor recebe *warnings* faz *broadcast* dos mesmos para todos os veículos, pois é informação útil que deve ser partilhada entre todos os veículos na estrada.

Resumindo, as entidade que constituem o sistema são:

1. **Cliente,** que usa o contato ControlService, permitindo a um cliente usar o serviço, esta é denominada como a aplicação cliente, podendo haver várias instâncias da mesma em simultâneo.
2. **ControlServer,** é a entidade servidora que implementa o contrato ControlService e é responsável por fornecer o serviço descrito no contrato.
3. **CentralServer,** é a entidade já existente que usa o contrato **CentralService** via gRpc, esta entidade simula uma entidade bancária, como dito anteriormente, sendo que é utilizada apenas pela entidade **ControlServer** quando é necessário pedir o montante que o cliente tem de pagar.

Portanto, um fluxo normal de um cliente será:

1. Entrar na via, para tal executa o chamada **Enter** ao servidor de controlo (ContolServer);

```
ControlServiceGrpc.newBlockingStub(channel).withDeadlineAfter( duration: 5, TimeUnit.MINUTES).enter(initial);
```

Do lado do servidor, o pedido do cliente é registado num mapa, simulando uma base de dados, para que se possam ser feitas validações posteriormente.

2. O cliente executa a chamada **Warning**, registrando automaticamente em ambas instâncias, cliente e servidor, os stubs (StreamObservers) pelos quais usaram para comunicar os warnings. Esta call não constitui do envio do warning em si.Registo do stubs na imagem em baixo e implementação do *WarningObserver* (StreamObserver do objeto Warning).

```
StreamObserver<WarnMsg> warningObserver = ControlServiceGrpc.newStub(channel).warning(new WarningObserver());
```

```
public class WarningObserver implements StreamObserver <WarnMsg> {

    public AtomicBoolean isCompleted = new AtomicBoolean( initialValue: false);

    @Override
    public void onNext(WarnMsg warnMsg) { System.out.println("ATTENTION! -> " + warnMsg.getWarning()); }

    @Override
    public void onError(Throwable throwable) {
        System.out.println("Server crashed " + throwable);
    }

    @Override
    public void onCompleted() {
        isCompleted.set(true);
        System.out.println("Server called on complete");
    }
}
```

Do lado do servidor, quando é feito o registo do stub, o servidor guarda o StreamObserver numa instância única de um objeto pelo qual denominamos como `MessageBroadCast`, sendo este responsável por guardar todos os StreamObservers dos clientes, sendo este responsável por gerar um identificador único por cada stub agnóstico ao cliente.

3. O cliente têm a possibilidade de enviar um *warning* para o servidor que será posteriormente *broadcasted* para todos os clientes “subscritos” com um stub

(StreamObservers) do lado do ContolServer. Envio de um warning por parte de um cliente para o ContolServer, na imagem abaixo.

```
warningObserver.onNext(WarnMsg.newBuilder().setId(ID).setWarning("rocks in the middle of the road").build())
```

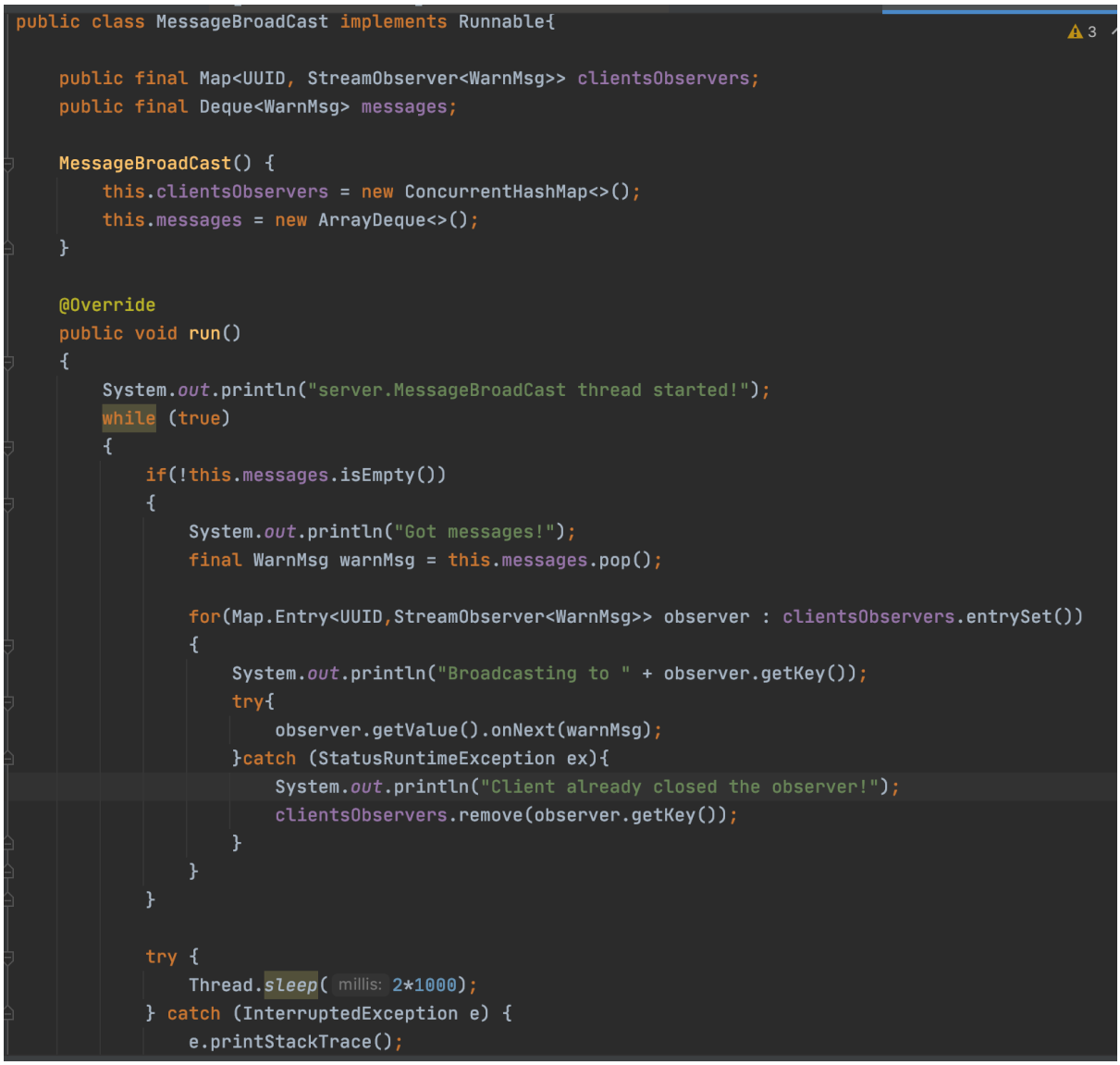
Do lado do servidor, quando este recebe um *warning,* o warning é registado na numa lista única de warnings na instância de `MessageBroadCast`.

Simultaneamente, na instância de `MessageBroadCast` existe uma *thread* que observa a lista de *warnings* sendo que quando existe uma mensagem na lista, esta é broadcasted para todos os StreamObservers registados.

4. O cliente recebe automaticamente a qualquer altura, desde que esteja ainda dentro do trajeto, todos os *warnings* que o ContolServer emitir.
5. Por fim, o cliente executa a call **Leave** para o ContolServer, neste momento o servidor executa um pedido ao CentralServer pelo montante de pagamento que o cliente tem de pagar, retornando este montante ao cliente. O cliente é também removido do mapa de cliente ativos.
   Na nossa implementação, é neste momento que o cliente deve executar o *onCompleted* para que quando recebido pelo servidor este seja removido dos StreamObservers.

Foi também implementado a funcionalidade `.withDeadlineAfter` a cada stub (cliente e servidor) de forma a obter um *timeout* em cada pedido, assim caso o *timeout* seja atingido uma exceção é lançada e tratada pelas entidades responsáveis.

Servidor



```java
public class MessageBroadCast implements Runnable{

    public final Map<UUID, StreamObserver<WarnMsg>> clientsObservers;
    public final Deque<WarnMsg> messages;

    MessageBroadCast() {
        this.clientsObservers = new ConcurrentHashMap<>();
        this.messages = new ArrayDeque<>();
    }

    @Override
    public void run()
    {
        System.out.println("server.MessageBroadCast thread started!");
        while (true)
        {
            if(!this.messages.isEmpty())
            {
                System.out.println("Got messages!");
                final WarnMsg warnMsg = this.messages.pop();

                for(Map.Entry<UUID,StreamObserver<WarnMsg>> observer : clientsObservers.entrySet())
                {
                    System.out.println("Broadcasting to " + observer.getKey());
                    try{
                        observer.getValue().onNext(warnMsg);
                    }catch (StatusRuntimeException ex){
                        System.out.println("Client already closed the observer!");
                        clientsObservers.remove(observer.getKey());
                    }
                }
            }

            try {
                Thread.sleep( millis: 2*1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

MessageBroadcast serviço implementado pelo servidor responsável pelo envio dos warnings

```java
public class ServerObserver implements StreamObserver<WarnMsg> {

    private final StreamObserver<WarnMsg> clientObserver;
    public final UUID observerId;

    public ServerObserver(final StreamObserver<WarnMsg> clientObserver) {
        this.clientObserver = clientObserver;
        this.observerId = UUID.randomUUID();
    }

    @Override
    public void onNext(WarnMsg warnMsg)
    {
        //System.out.println("New warning received! " + warnMsg + " on observerId " + this.observerId)
        Server.messageBroadCast.messages.add(warnMsg);
        //Syste                                                    messageBroadCast.messages.size());
    }

    @Override
    public void onError(Throwable throwable) {

    }

    @Override
    public void onCompleted() {
        System.out.println("Observer " + observerId + " called onCompleted!");
        Server.messageBroadCast.clientsObservers.remove(observerId);
        this.clientObserver.onCompleted();
    }
}
```

server.Server
public static MessageBroadCast messageBroadCast

StreamObserver implementado pelo servidor e enviado ao cliente

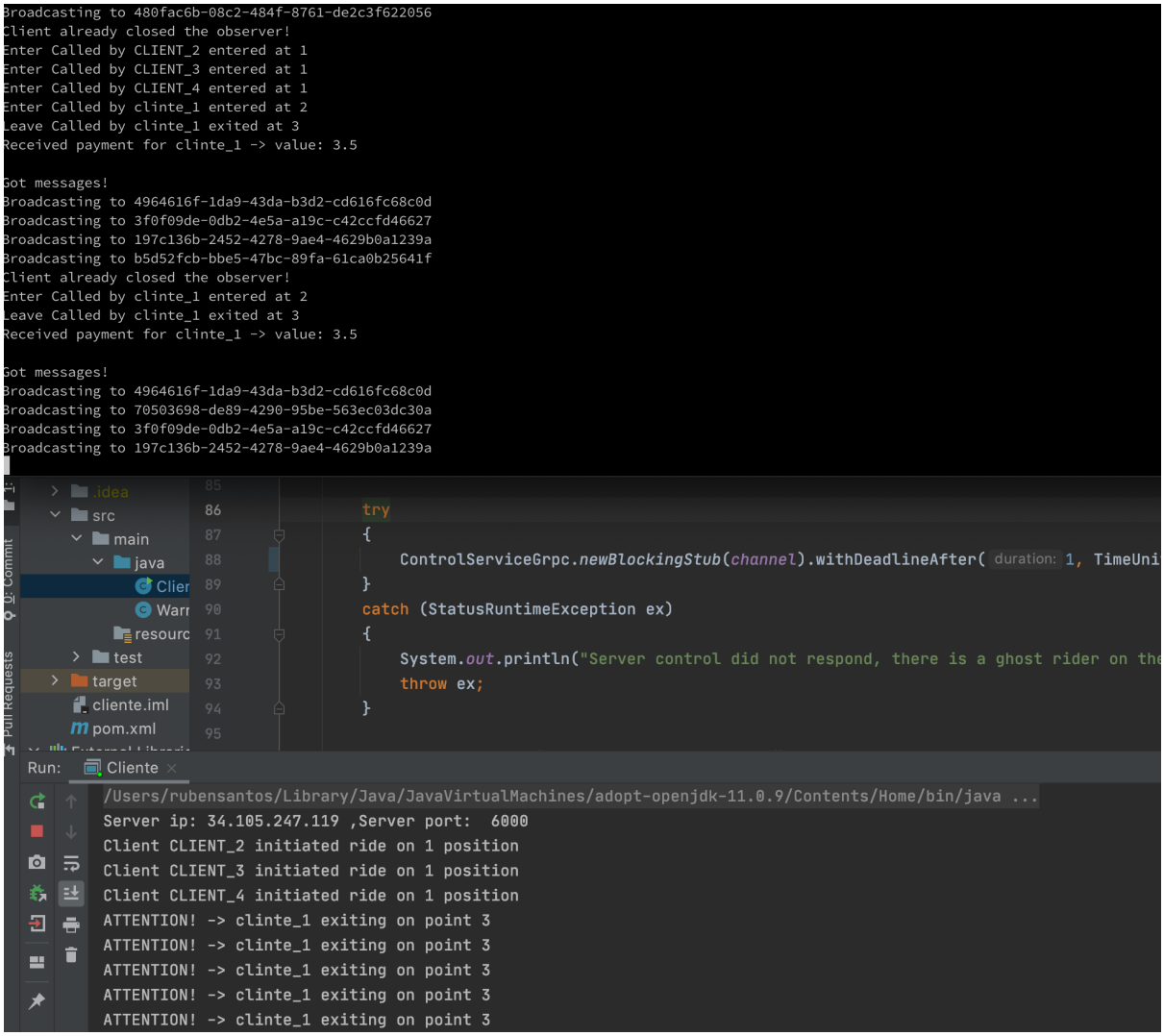
Cliente

```java
static String svcIP="34.105.247.119";
static int svcPort=6000;

static ManagedChannel channel;

public static void main(String[] args) {

    if(args.length > 0){
        svcIP = args[0].isEmpty() ? svcIP : args[0];
        svcPort = args[1].isEmpty() ? svcPort : Integer.parseInt( args[1] );
    }
    System.out.println("Server ip: " + svcIP + " ,Server port:  " + svcPort);

    channel = ManagedChannelBuilder
            .forAddress(svcIP, svcPort)
            .usePlaintext().build();
```

Criação do channel Cliente ao ControlServer (similar ao ControlServer <-> CentralServer)

```
Server ip: 34.105.247.119 ,Server port:  6000
Client 6543 initiated ride on 1 position
Client 123456 initiated ride on 2 position
Sending warning: Danger on the road 123456 entered on the 2 entry
ATTENTION! -> rocks in the middle of the road
ATTENTION! -> rocks in the middle of the road
Sending warning: 123456 exiting on point 3
Sending warning: 123456 payed value: 3.5


Process finished with exit code 0
```

Output consola do Cliente



Na imagem a cima podemos ver os logs do servidor e de 3 clientes, é feito o registo de 3 clientes que não saem do trajeto, através de outro cliente do computador do meu colega é feito um fluxo normal de um cliente, pertendendo demonstrar aqui o broadcast da mensagem enviada pelo cliente 1 para o cliente 2, 3 e 4.