Università degli Studi
di Napoli Parthenope

Reti di calcolatori e Laboratorio di reti di calcolatori
Anno 2023/2024

Progetto Starship

Vincenzo De Candia
0124002539

Università degli Studi di Napoli Parthenope

Indice

1 Descrizione del progetto

Il progetto si focalizza sullo sviluppo di un sistema per consentire la navigazione sicura di una navicella spaziale attraverso una tempesta di meteoriti. La navicella agisce da client, mentre un server dedicato gestisce il settore spaziale dei meteoriti. All'avvio, il client invia le dimensioni della mappa al server. Successivamente, il server genera ogni 2 secondi n pacchetti UDP corrispondenti alla posizione dei meteoriti generata in modo casuale. La navicella deve evitare i meteoriti generati dal server, e nel caso in cui si trovino nella stessa posizione, il client riceve un alert e ha 3 secondi per eseguire una manovra per evitare la collisione. La comunicazione avviene tramite il protocollo UDP (User Datagram Protocol), che offre una trasmissione veloce e non affidabile dei dati.

2 Descrizione e schema dell'architettura

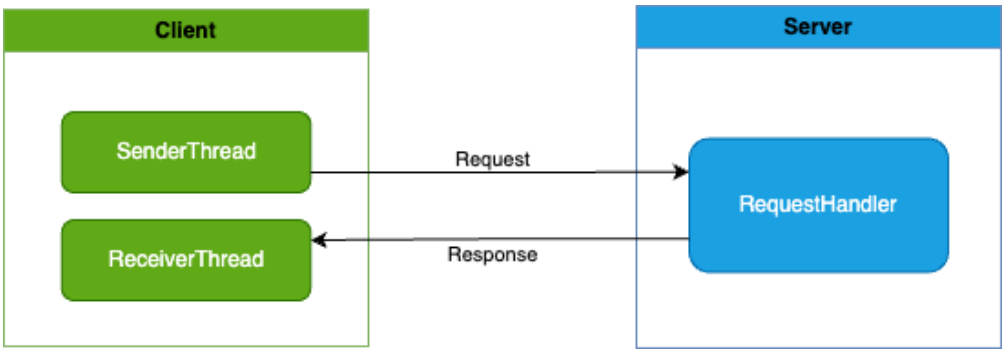

Figura 1: Architettura client-server

2.1 Server

- Il server UDP si mette in ascolto su una porta specifica per ricevere pacchetti dai client.
- Quando riceve un pacchetto, crea un nuovo thread di gestione della richiesta (RequestHandler) per gestire la richiesta del client.
- Ogni thread di gestione della richiesta (RequestHandler) si occupa di ricevere i dati dal client, elaborarli e inviare eventuali risposte.

2.2 Client

- Il client UDP utilizza due thread separati: uno per inviare dati al server (SenderThread) e uno per ricevere dati dal server (ReceiverThread).
- Il thread SenderThread si occupa di inviare dati al server, mentre il thread ReceiverThread si occupa di ricevere dati dal server.
- Entrambi i thread sono eseguiti contemporaneamente per consentire al client di gestire l'invio e la ricezione di dati in modo concorrente.

3 Dettagli implementativi dei client/server

3.1 ServerUDP

Il server crea un'istanza di **DatagramSocket** associata a una porta specifica per ascoltare le richieste in arrivo dai client. Il server UDP ascolta continuamente per i pacchetti in arrivo utilizzando il metodo *receive()* della classe **DatagramSocket**. Quando arriva un pacchetto, il server istanzia un nuovo oggetto **RequestHandler** in un thread separato per gestire la richiesta del client. Questo consente al server di continuare ad ascoltare per ulteriori richieste mentre gestisce quelle esistenti. **RequestHandler** elabora i dati ricevuti dal client, determinando il tipo di messaggio ricevuto tramite la classe **JSONHandler**. Se il messaggio è "mapDimension", il gestore di richieste imposta la larghezza della mappa del gioco e imposta la variabile booleana *TRUE* per permettere al ciclo while di essere eseguito ed inviare dati al client. Se il messaggio è "game over", il gestore di richieste interrompe il thread di invio. I dati vengono incapsulati in pacchetti Datagram e inviati al client tramite il metodo *send()* del socket.

3.2 ClientUDP

All'avvio del gioco, viene istanziata la classe **UDPClient**. Nel suo costruttore, vengono inizializzati i parametri del server insieme all'oggetto **EnemySubject**. Quest'ultimo è responsabile della gestione degli oggetti nemici all'interno del gioco, consentendo al client di ricevere informazioni aggiornate sugli avversari e di aggiornare lo stato del gioco di conseguenza.

Durante l'inizializzazione, viene creato un socket Datagram per consentire la comunicazione con il server. Inoltre, vengono avviati due thread: **senderThread** e **receiverThread**.

Il thread **senderThread** ha il compito di inviare messaggi al server tramite pacchetti Datagram. Questi messaggi sono rappresentati come oggetti JSON e possono includere informazioni cruciali per il funzionamento del gioco, prima di essere inviati vengono convertiti in stringhe. In particolare, il client può inviare solo due tipi di messaggi: il primo comprende le dimensioni della mappa di gioco, mentre il secondo contiene un messaggio di "game over" per segnalare la fine della partita.

Il thread **receiverThread**, invece, è responsabile della ricezione dei dati inviati dal server. Questi dati includono informazioni sugli oggetti nemici presenti nel gioco, come la loro posizione e lo stato attuale. Una volta ricevuti, questi dati vengono elaborati e utilizzati per aggiornare lo stato del gioco, consentendo al client di reagire di conseguenza.

4 Parti rilevanti del codice sviluppato

```

/* Il server UDP utilizza un thread separato per gestire ogni richiesta attiva
    proveniente dai client. Quando il server riceve una nuova richiesta da un
    client, controlla se quel client ha gi  una richiesta attiva. Se il server
    trova una richiesta attiva associata al client, interrompe il thread associato
    a quella richiesta precedente. Questo viene fatto per evitare conflitti e
    sovrapposizioni nel processamento delle richieste da parte dello stesso client.
     Successivamente, il server istanzia un nuovo oggetto per gestire la nuova
    richiesta e lo memorizza in un HashMap, associandolo al client che ha inviato
    la richiesta. */

byte[] receiveBuffer = new byte[1024];
DatagramPacket receivePacket = new DatagramPacket(receiveBuffer, receiveBuffer.
    length);
socket.receive(receivePacket);

InetAddress clientAddress = receivePacket.getAddress(); // Get the client's
    address
int clientPort = receivePacket.getPort();

String clientString = clientAddress.toString() + clientPort;

if (clientThreads.containsKey(clientString)) {
    Thread previousThread = clientThreads.get(clientString);
    if (!previousThread.isInterrupted()) {
        previousThread.interrupt();
    }
}

RequestHandler requestHandler = new RequestHandler(receivePacket, socket);
clientThreads.put(clientString, requestHandler);
requestHandler.start();
```

Listing 1: ServerUDP

```
/*
Il thread del server UDP, una volta avviato, esegue il metodo receive() per
    elaborare il pacchetto ricevuto dal client. Se il server riceve un pacchetto
    contenente le dimensioni della mappa, aggiorna una variabile di stato (running)
     a true. Questo segnala al thread di avviare un ciclo while successivo, che
    consentir  al server di inviare pacchetti contenenti posizioni casuali di
    meteoriti al client ogni 2 secondi.
*/
@Override
    public void run() {
        receive(); // Receive and process data from the client
        if (running) {
            try {
                while (!Thread.currentThread().isInterrupted()) {
                    // Generate random positions for enemy entities
                    for (int i = 0; i <= numberPacket; i++) {
                        Random random = new Random();
                        int randomValue = random.nextInt(getDimensionWidth());
                        JSONObject enemy = new JSONObject();
                        enemy.put("positionX", randomValue);
                        enemy.put("positionY", 0);
                        send(enemy); // Send the position of enemy to the client
```

```
                    }
                    Thread.sleep(2000); // Pause for 2 seconds before sending the
                        next batch of positions
                }
            } catch (InterruptedException | IOException e) {
                System.out.println("Thread interrupted");
            }
        }
    }

    private void receive() {
        JSONObject jsonObject = new JSONHandler(receivePacket).getJsonObject();
        String typeMessage = jsonObject.getString("message");
        switch (typeMessage) {
            case "mapDimension":
                // Set the dimension width and allow the sender thread to start
                dimensionWidth = jsonObject.getInt("width");
                running = true;
                break;
            case "game over":
                // Received game over message, stop the sender thread
                System.out.println("STOP");
                running = false;
                break;
            default:
                break;
        }
    }

    public void send(JSONObject jsonObject) throws IOException {
        InetAddress clientAddress = receivePacket.getAddress();
        int clientPort = receivePacket.getPort();
        DatagramPacket sendPacket = new JSONHandler(jsonObject, clientAddress,
            clientPort).getDatagramPacket();
        socket.send(sendPacket); // Send the packet to the client
    }
```

Listing 2: Request handler

```
try {
            while (!Thread.currentThread().isInterrupted()) {
                if (thereIsDataToSend()) {
                    JSONObject jsonObject;
                    jsonObject = getJSON();
                    DatagramPacket datagramPacket = new JSONHandler(jsonObject,
                        inetAddress, serverPort).getDatagramPacket();
                    socket.send(datagramPacket);
                }
            }
        } catch (IOException e) {
            System.err.println("Error while sending data: " + e.getMessage());
            Thread.currentThread().interrupt();
        } finally {
            socket.close();
        }
```

Listing 3: Client SenderThread

```
@Override
```

```
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                List<AObject> enemyList = new ArrayList<>();
                enemyList.add(receive());
                enemySubject.setStateEnemyList(enemyList);
            }
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Receiver closed");
        }
    }

    public AObject receive() throws IOException, ClassNotFoundException {
        byte[] receiveBuffer = new byte[1024];
        DatagramPacket receivedPacket = new DatagramPacket(receiveBuffer,
            receiveBuffer.length);
        socket.receive(receivedPacket);
        JSONObject jsonObject = new JSONHandler(receivedPacket).getJsonObject();
        int x = jsonObject.getInt("positionX");
        int y = jsonObject.getInt("positionY");
        return new Enemy(x, y);
    }
```

Listing 4: Client ReceiverThread

5 Manuale utente con le istruzioni su compilazione ed esecuzione

5.1 Installazione di Maven su MacOS

Prima di installare Maven bisogna installare homebrew

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/HEAD/install.sh)"
```

Una volta installato Homebrew, è possibile installare Maven eseguendo il seguente comando:

```
brew install maven
```

Dopo l'installazione, verifichiamo che Maven sia stato installato correttamente eseguendo:

```
mvn -version
```

5.2 Installazione di Maven su Linux

Su Linux, si utilizza il gestore di pacchetti apt per installare Maven. Ecco come farlo:

```
sudo apt update
sudo apt install maven
```

Dopo l'installazione, verifichiamo che Maven sia stato installato correttamente eseguendo:

```
mvn -version
```

5.3 Compilazione ed esecuzione del client

Per compilare ed eseguire il client eseguire il seguente comando:

```
cd Starship
mvn clean javafx:run
```

5.4 Compilazione ed esecuzione del server

Per compilare il server eseguire il seguente comando:

```
javac -cp ServerUDP/lib/json-20200518.jar:.  ServerUDP/src/*.java
```

Per avviare il server eseguire i seguenti comandi:

```
cd ServerUDP/src
java -cp ../lib/json-20200518.jar:. Main
```