Multithreaded Battle City Console Game

5/07/2020

Students:

ALEGRE IBEZ, Vctor Augusto 20130504C
ZAVALETA BUENO, Romel Rolando 20120236F
ZEVALLOS LABARTHE, Enrique Martn 20130384H
Universidad Nacional de Ingeniería, Facultad de Ciencias,
e-mail: victoralegre@uni.pe, romelzavaleta@uni.pe, enrique.zevallos.l@uni.pe

Subject:

CC462 - Concurrent & Distributed Systems
Lab 2

Abstract

In this laboratory, we will develop the well-known game *Battle City* implementing a client-server model. We opted for a distributed and concurrent approach where both server and client implement threads to optimize performance. The server will continuously listen for new connections and will be in charge of most of the processing needed. Clients will only have to worry about the GUI and after establishing the connection with the server, send keystrokes and receive the data needed to re-draw the map and the tanks (other clients and themselves) within.

Keywords: Threads, Concurrency, Distributed, Console, Game, Battle, City.

Contents

1 Introduction

The notion of client-server models has been used extensively throughout the gaming industry. This allows distributed processing of the payload, such that the client will only have to execute a lightweight version of the program. The vast majority of the processing is done by the server. Clients are only in charge of sending orders for the server to execute, and creating and updating the GUI. In this particular instance, we have developed the logic of the game within a server, to which several clients will connect to play. The server is in charge of updating the map with the information it receives from the clients. Clients will conect to the server and send keystrokes in order to shoot and move within the map. The server will then return the updated map for the client to draw. This way, the client can run in a wider variety of hardware choices with similar performance.

2 Theoretical Framework

For the implementation of the game, we are using a multithreaded approach in both the client and the server. The server will use a thread for each new player that asks to connect. Its implementation consists of two infinite loops. One of them waits for new connections, and the other sends updates (in 100 millisecond intervals) of the players' positions, shots and collisions. Players are objects stored in a list and instantiated using a network socket. This socket will provide both input and output streams for the client keystrokes and map characters, respectively. Additionally, players will have a list of *bullet* objects. Both players and bullets will have attributes of position in a 2D "battle ground". The instantiated server will iterate over all player objects to determine their direction and movements. If the coordinates of a bullet are the same as those of a player, then this player will be eliminated. For the client, only two threads are implemented. The main thread is in charge of the GUI and event handling from keystrokes. The other thread will read the output from the server, either map updates or an "eliminated" notification.

3 Methodolody

3.1 The Server

The server implementation is made in our main class, which instantiates the *ServerThread* class in a *server* object, and calls its methods *start* and *enviarUpdates*. This class inherits from the *Thread* class and so it implements a runnable method when *start* is called. In this case, we overrode the *run* method to run an infinite loop and wait for new connections. After establishing a new connection, a player object named *jugador* is instantiated using the class *ServerJugador* which also inherits from the *Thread* class. The spawning position of the new object is generated randomly. This new object is added to the list *jugadores* and then *jugador.start()* will call its runnable method. The *run* method is overridden to establish two data streams, input and output. We use a *while* loop that checks if the *running* attribute is still *true*. This value is modified to be set as *false* when the *notificaPerdido* method is called, after the player object (*jugador*) is added to the *gamer_eliminado* list, when its position is the same as that of a bullet. This loop keeps on reading the input stream if *running* is set to *true*, as well as establishing the object's movement and determining if shots were fired. If the *key* it receives from the input stream is *'x'*, then it will instantiate a bullet (*bala*) object. This bullet will receive its initial coordinates from the player that instantiated it, its state is set to true and it is added to the *balas* list of objects. Another method of the *ServerJugador* class is *enviar*, which sends a message in UTF-8 encoding through the output stream. This method will be called from the *server* object to send the updated map to the client. After the player is instantiated and initialized in the server's runnable method, the method *enviarUpdates* is called. This method creates a thread and calls its runnable method where an infinite loop updates the total amount of players and creates a list of eliminated players. It sleeps for 100 milliseconds, then prints the number of players and coordinates of a collision. If there were no collisions, the coordinates will be (0,0). Then a *for* loop iterates over the list of players and for each one of them, it will call the *CoreJuego* method which modifies the orientation and position based on the *direccion* attribute of the player. It will also check for bullets, determining its movements and collisions. If the object it collides with is a 'tank', the collision coordinates are modified and thus the condition in the next line of code will verify. After the player is passed to the method *CoreJuego*, and the physics of the game is implemented, we use an *if* statement to determine if a bullet collided with a 'tank', and proceed to add said player to the *gamer_eliminado* list of objects. The collision coordinates are then set back to (0,0). Then, after exiting the if condition the method *enviar* is called for each player, and the *for* loop finishes. Finally, we iterate over the *gamer_eliminado* list and call the method *notificaPerdio* to send the notification *"pierdes"* through the output stream, and set the *running* attribute of the player to *false*. All disqualified players are remove and the thread sleeps for an additional 100 milliseconds. The last class we ought to mention is *CargarMapa*. This class will create a 16x32 matrix reading each string of the *mapa.txt* file. If any exception should be thrown, the appropriate *try* and *catch* statements are in place. A *while* and a *for* loop are implemented to iterate over the whole text file and populate the map matrix.

3.2 The Client

4 Results and Discussion

4.1 Calculating Pi

4.2 MergeSort

In the example we implemented (found in our GitHub Repository), we

5 Conclusions

The implementation of multithreading will allow us to improve time efficiency in our algorithms if it is implemented correctly. If we were to use more threads than physically available (virtual threads), this will not increase the performance. Instead it increases cost of parallelizing which will in turn result in a diminished performance.

6 Code

The code is in the following link `https://github.com/ezevallos/CC462_EjemplosConcurrencia`
