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

The 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. 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 loop keeps on reading the input stream if so, 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.

3.1 Calculating Pi

We ought to calculate the irrational number Pi using an algorithm acquainted in the early XVIII century by English mathematician John Machin. This algorithm first calculates $\pi/4 = 4*arctan(1/5) - arctan(1/239)$. (FRIESEN, 2015)[**?**] To calculate the arctangent of an angle in radians, we use Taylor Series, where

$$arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9}...$$

To execute the code concurrently, we built a method named *arctan*, that will instantiate *nHilos* threads. Each thread will compute a subtotal through *runnable* object *ArctanThread* that is in charge of calculating the Taylor Series. We then implement a barrier to checkpoint the threads, utilizing the *join()* method and proceed to add up the subtotals. Finally, we multiply this total by 4 to obtain the value of π.

4 Results and Discussion

4.1 Calculating Pi

The benefits of implementing a multithreaded approach becomes quite evident. Just looking at our example execution (found in our GitHub Repository) where we attempt to obtain π to 100, 000 digits, when using four threads, our execution time is decreased to 27 % of the total time it took to execute using only one thread. Furthermore, Figure 1 demonstrates that as we continue to increment the amount of decimal places for pi, and the amount of threads used to execute the algorithm, its asymptotic analysis resembles that of O(n).

4.2 MergeSort

MergeSort is based on a 'divide and conquer' technique, implemented by John Von Neumann in the year 1945. The algorithm is the following:

Within our code, step 1 and 2 are found in recursive method *MergeSort* belonging to *Ordenador* class. Step 3 is found within the *merge* method, in the same class. The last step is found within the *finalmerge* method in the *MergeSortHilos* class.

In the example we implemented (found in our GitHub Repository), we used two threads defined within the *main* method. The code is hardcoded for a set of numbers within the range of 0 to 1000, and these are generated randomly. The ordered list will be printed in console, line by line, followed by the execution time. **Note: A bigger set can be sorted. For this we advise commenting the print function, so as to limit the buffer overflow.** We executed a *MergeSort* for a 4,000,000 data set in 528 miliseconds.

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`