Final Report

Pedro Lopes, Bruno Santos, Afonso Caetano
81988, 82053, 82539
Instituto Superior Técnico
Desenvolvimento de Aplicações Distribudas
pedro.c.lopes@tecnico.ulisboa.pt, bruno.o.santos@tecnico.ulisboa.pt, afonso.caetano@tecnico.ulisboa.pt

Abstract

The DAD project consists of implementing a simplified fault-tolerant real-time distributed gaming platform.

1. Introduction

In the beggining of the semester, students were proposed to develop a project, which consists of a fault-tolerant on-line gaming platform. The game, an on-line version of the Pacman, is based on a server-client model and has two modules: a Client module, which is a Windows Form that offers a User Interface to both the gaming zone and chat to the user; and a Server module, that could be implemented using a console or a Windows Form as well, which is responsible for maintaining the state of the game and synchronize all the actions submitted by the clients. There is also the PuppetMaster, which role is to provide a centralized component that can control experiments to the project.
This report is divided into four sections: Section 2, "Design", where we explain the design of our project, from the Server module to the PuppetMaster, explaining how we implemented each and the non implemented features; Section 3, "Qualitative and quatitative evaluation", in which we make a qualitative and a quantitative evaluation of our project; and a Section 4, "Conclusion", where we debate what was accomplished and what could have be done in order to make the project better.

2. Design

2.1. Server

The Server can be started manually or using the PuppetMaster. When manually started, it is defined a game rate (frequency of the game updates) and the number of players that the Server needs to have registered in order to start the game. Every game rate time, it will process all the movements received from the clients and send back to them the result of that computation. During that, the server calculates the new positions of the ghosts, the new positions of each pacman (client), the new score of each pacman in case they eat a coin and removing the corresponding coin. It also sees if a pacman hits a ghost or a wall, causing that client to stop, sending the message "GAME OVER". In the case where all coins have been eaten, the Server sends a message "VICTORY" to the clients, stopping them.

2.2. Client

Like the Server, the Client can also be started manually or through the PuppetMaster. The client also has a game rate and, every game rate time, it sends it's movements to the Server. When the Client receives a message from the Server with the update of the game state, it makes changes according to the Server's message, displaying only the coins that the Server sends, the positions of the ghosts and pacmans. If it receives the message "GAME OVER" (represented by -1 in the code), the Client stops sending messages to the server with it's moves. On the other hand, if the Client receives the message "VICTORY" (represented by -2 in the code), it stops sending messages to the Server with it's moves.

On the client side it was also implemented a peer-to-peer chat where each one of them add a list containing all the clients in a game session. Every time a client wants to send a message, all the other clients in the game session receive that message, which means that it isnt possible for a client A talk only with B if the game is being played for A, B and C. C will always see what A only wanted to send to B.

Although the chat presents two possible problems when used in a distributed system, which both consist on the delay/not reception of the messages, leading to different chat displays for each client when they all should have the same messages presented in the same order.

The first one applies on the cases where A has 1, 2, 3,

being 1, 2 and 3 messages, B and C should have 1, 2, 3 as well, instead of 1, 3, 2 or any other possible order. The second one applies on the cases where the messages dont arrive to all clients, for example A sends 1 but only A and B receive it, C never gets to receive it because the connection was down in the message transmission moment. To prevent those errors to happen it was implemented the vector clock, which is a vector initialized with the size of the number of clients, with all positions values equal to zero. Every time a client sends a message it will add one to its position and send the vector along the message. When a client receives a message, compares its vector with the received one and if all the values are bigger than the received one, or if they are all bigger except for one, which is equal, or if they are all the same except for one, which is equal the received minus one, the message is displayed. Otherwise the message is placed on hold alongside the vector and the time when the client received it. Every x seconds the Client sees if a message is on hold for more than a minute, and in case it is, it asks to all the clients if they have the specified vector displayed. If they do, they send the previous message and corresponding vector. When the Client receives it, it will do the process referred above, and if the message is displayed, it will see if any message is on hold, depending on that one.

2.3. PuppetMaster

The role of the PuppetMaster process is to provide a platform where it is possible to control experiments, facilitating the execution of tests to the system, such as the activation of clients/servers and orchestrating their execution. For this, the PuppetMaster was develop being a Windows Form with two TextBox: one of them is used to write commands (being every command processed in a differently way, depending on the first word of the command, that defines the desired instruction to be executed), the other is an output TextBox where the results of some particular commands are shown.

In our project, the PuppetMaster consists of a simple class that uses both Client and Server classes as reference so it is able to call methods defined in them.
As mentioned before, the PuppetMaster is used to activate clients and servers. This means that Client and Server can be launched through the PuppetMaster without the need of being launched manually. If the command ”StartClient PID PCS_URL CLIENT_URL MSEC_PER_ ROUND NUM_PLAYERS [filename]” is given to the PuppetMaster, a Client is executed and the arguments CLIENT_URL, MSEC_PER_ ROUND and NUM_PLAYERS are passed to the creation of the Client, being its services exposed at a port that is specified on this URL and having the milliseconds per round and number of players desired by the client defined by the last two arguments. Similarly, if the command ”StartServer PID PCS_URL SERVER_URL MSEC_PER_ROUND NUM_PLAYERS” is given, a Server is executed following the same logic as the client, which means that the server services will be exposed on a port specified by the SERVER_URL and the milliseconds per round and number of players desired are defined by the arguments MSEC_PER_ ROUND and NUM_PLAYERS (passed as arguments on the creation of the server). In both this cases, on the PuppetMaster, there is a dictionary that associates the PID argument with the CLIENT_URL/SERVER_URL, so we can keep track on what each PID refers to. Every time you activate a server or a client there is also an addiction to the respective URL to a list of active servers or active clients, depending of what the PuppetMaster activated.

Having a dictionary with PID and URL relations, the PuppetMaster can execute other operations on client and server processes through the commands ”Crash PID”, ”Freeze PID” and ”Unfreeze PID”. For each of this commands, as they represent a specific action on a specific PID, we first analyze if the given PID corresponds to a server or a client through the analysis of the dictionary early mentioned and, depending on the result, the PuppetMaster creates an instance of a remote interface (IClient if it corresponds to a client or IServer if it corresponds to a server) specified by the port in the URL that corresponds to the given PID, so it is able to remotely call functions offered by this services.

If the command executed in the PuppetMaster is ”Crash PID”, the PuppetMaster calls a funcion (getProcessToCrash()), offered by the services, that crashes the desired process.
Both the commands ”Freeze PID” and ”Unfeeze PID” had the need to modify the server and client architecture in order to have a flag (freeze) that is used specifically to these commands. This flag, when false, allows the game to work normally, with the client sending the movements to the server and the server receiving this messages, computing the logic and sending the update messages. However, when the flag is true, the client shouldn’t be able to send the pretended movement to the server nor receive messages from the server to update the game and the server shouldn’t be able to receive messages from the client nor compute the logic of the game. So, in the PuppetMaster, when this commands are call, it looks on the dictionary for the URL associated with the PID passed as argument, creates an instance of the remote interface type accordingly, and calls either the setFreeze() method (corresponding to the ”Freeze PID” command) or the setUnfreeze() method (corresponding to the ”Unfreeze PID” command) present in both IClient and IServer interfaces, in order to set this flag to true or false, respectively.

The PuppetMaster can also execute the ”GameStatus” command, that is responsible to print to the output TextBox information about the state of the system, that is which

entities are running and which are presumed to be down. For this command, once again, the PuppetMaster needs to create instances of the remote interfaces. The only difference is that, this time, since this is a command regarding the global state of the system, there must be an instance for every server and client that were added to the active servers and active clients list. For every instance created, the PuppetMaster calls a method (getStatus(), implemented in both IClient and IServer interfaces) that basically returns the value 1 and, depending of receiving this value or not, the corresponding URL (present in the list) is added to a string of who is running (the ones that return the value 1) or to a string of who is presumed to be down (the ones that, for being supposedly down, don't return the value 1).
For the command "Wait x_ms", we used the System.Threading.Thread.Sleep() function.

2.3.1 Non implemented features

There were some functionalities of the PuppetMaster that weren't fully implemented or even at all.

The "StartClient", "StartServer" and "GameStatus" are three commands that weren't fully implemented:

- The first one is supposed to be able to support a sixth argument, [filename], which our implementation of the PuppetMaster doesn't support. This could have been easily fixed by creating a function that read a document and executed the commands accordingly to what was read from the file:

- The Server and Client creation commands are supposed to create a client and a server that expose their services on the URL given by SERVER_URL and CLIENT_URL, respectively. This wasn't fully implemented as only the port is being used to the address where the service is being provided. This could have been easily fixed by using the whole URL to the service instead of using "tcp://localhost" + port (given in the URL provided to the PuppetMaster);

- The third commands works for all the running processes but crashes every time it tries to run the method getStatus() on a shutdown process, since there isn't any treatment of exceptions for when the process doesn't answer to the method call. Since the process is assumed down, we can never remotely call a method on it, as it will throw an exception, which we don't treat. This could have been fixed by adding a try-catch block to when we assume that the process that is shutdown will not able to respond to the method call and, on the catch block, immediately add it to the string of who is presumed to be down.

Now, about was wasn't fully implemented, we can start by saying that the Process Creation Service (PCS) wasn't implemented. For this we would have needed a new class (PCS) and a new remote interface (IPCS) that the PuppetMaster should create an instance of. If there is any argument that doesn't match what the PuppetMaster is expecting, it will crash, as we didn't add exceptions list for this, but could have been fixed if we did (for example, if we give the command "Freeze PID" without the PID argument, the program will crash as he was expecting an argument).

The commands that weren't implemented in our PuppetMaster are the ones next described:

- "InjectDelay src_ PID dst_PID", since our whole project runs in a synchronous mode, which means that both Client and Server need to have the exact game speed to work normally. If the timer on the Client is smaller that the one at the Server, there would be a lot of exceptions thrown, but if the timer on the Server is smaller than in the Client, although the project doesn't crash, there would be some movements that would be lost;

- "LocalState PID round_id" wasn't implemented as we didn't implement a round id mechanism on our project. If we did have this identifier on the Server and Client, we could easily get the status of the game as we only needed to look for that id the corresponding PID.

2.4. Fault Tolerance

The fault tolerance paradigm in DAD-OGP can be divided in two categories since this project is a fault-tolerant on-line gaming platform, it is based on client-server communication hence fault tolerance should be guaranteed for both entities.

The game server is prepared to deal with several failures from the clients currently connected in the game session that can be divided in partial or complete clients failure.

In the case of a partial clients failure, where the Server is not able to send the updated information, it adopts a tolerant posture since for each update message it sends the Server monitors the number of failed attempts it has made in sending the command for each Client. In the case of 30 consecutive failures in sending the command, the Server adds the problematic Client to a "Down-Clients" list, where all the unresponsive clients are for a game session. After a Client has been added to this list from this point on the Client is declared "dead" to the server and no more updates of the status of the game are sent to the client in question. For the rest of the clients the game continues to run and this mechanism is completely invisible for them since the problematic client "pacman" is kept in the same position as before not influencing the game from this point on.

Note that the posture adopted by the server is quite tolerant since it needs 30 consecutive failures when sending the updated game status to a client to declare it ”dead”, so even if there are more than 30 failures in updating the client game status as long they are not consecutively the client will remain connected to the game session and the server will continue to keep it updated as the session unfolds.

In the case of the full clients failure where the server is not able to send the updated game status to all of the clients connected on a specific game session that can be triggered with the failure of all the clients at the same time or a consequent failure of clients until the number of failed clients is the same as the clients in the current game session. In both cases the server halts the game engine and also stops sending game status updates to the clients. In this case the game session must be restarted for all the clients in question.

2.4.1 Non implemented fault tolerance mechanisms

For the implementation of fault tolerance in the game server due to time constraints it was not possible to fully implement a complete replication mechanism of the server. The mechanism that was going to be adopted was supposed to support one server failure by having one backup replica running at the same time.

This mechanism would work by having the main game server sending periodic update messages with the current game status to the backup server, this way both games are identical in the main and the backup server. After a specific time period if the backup server stops receiving these messages from the main server it assumes that there was a failure and takes over the game session by binding to the port used by the main server and resuming the game where it was left. This is possible because at initialization the main server informs the backup server in which port it is operating and it gives all the information the backup server needs in order to take over the game at any time. These include the ”ip/port” of each client and the number of the clients currently connected.

3. Qualitative and quantitative evaluation

As seen in the section 2.4 the system supports some defensive mechanisms to prevent some faults, such as the one used in the case where a client goes down, the server continues running normally. While varying the number of players and server replicas involved in a gaming session, tested for a maximum of six players, it was possible to observe that the game throughput dont vary, this meaning that the observable throughput in the client is very similar if he is playing alone or in a multi-player session.

4. Conclusions

Overall it’s possible to identify that the basic game implementation was executed, allowing the game to be played with one or more clients remotely. The project is also resistant to some faults as seen in the section 2.4, although we don’t implement server replication, and we have a not fully implemented PuppetMaster.

The PuppetMaster implementation could have been fully implemented if the suggestion at the section 2.3.1 were implemented.

One big flaw is, as we didn’t implement server replication, that if the Server that is hosting the game crashes, the whole game is gonna be unplayable as the module responsible for the game processing is shutdown and there isn’t another Server that takes place in order to replace it and allow the normal game functioning.