DAD's Online Gaming Platform (**DAD-OGP**)

Desenvolvimento de Aplicações Distribuídas
Project - 2017-18 (IST/DAD): MEIC-A / MEIC-T / METI

October 13, 2017

Abstract

The DAD project aims at implementing a simplified (and therefore far from complete) fault-tolerant real-time distributed gaming platform.

1 Introduction

The goal of this project is to design and implement **DAD-OGP**, a fault-tolerant on-line gaming platform. The focus of the project is on the design and evaluation of mechanisms aimed at ensuring different consistency levels in presence of crash faults, rather than on the integration of advanced graphics libraries or the implementation of complex game logics (e.g., AI). In the light of the above observations, for simplicity, the **DAD-OGP** platform will be composed of two main components:

- a multi-player version of the PacMan game.
- a chat allowing communication among players involved in a game.

From an architectural perspective, the **DAD-OGP** platform is composed of two main modules:

- The **Client** module: a Windows form application that provides end users with access to both the gaming zone and the chat via a unified User Interface (UI).
- The **Server** module: the component responsible for maintaining the state of the on-line game and for synchronize the actions submitted in real-time by players. Students are free to opt for implementing the server side via either a Console application or a Windows form application.

The project shall be implemented using C# and .Net Remoting using Microsoft Visual Studio and the CLR runtime.

2 Server module

As already mentioned, the server is responsible for maintaining the state of the gaming sessions hosted by the **DAD-OGP** platform. In order to simplify implementation, the **DAD-OGP** platform is requested to host only a single type of game based on a variant of PacMan. Nonetheless, the design of the server side of the platform shall be as much as possible flexible and generic, in order to allow, potentially, its use also with other types of

games. In order to pursue this goal, the server-side module of the **DAD-OGP** platform models games as a deterministic state machines, which operate according to the following round-based approach:

1. wait for a fixed time period (`MSEC_PER_ROUND`) to gather inputs from clients. Inputs correspond to game's actions, e.g., a request for moving a player's PacMan along one direction.
2. compute updates to the game's state based on the inputs received for the current round, e.g., determining the updated positions of the monsters and PacMans.
3. send back to the clients the state changes for the current round. Each round can be assumed uniquely identified by a scalar timestamp.
4. move to the next round by increasing the corresponding timestamp and go to step 1, until the game is over.

The parameter `MSEC_PER_ROUND` is what regulates the actual game speed and represents an input parameter that has to be specified upon activation of the server module, along with the number of players in the game.

In addition to regulating the execution of the commands submitted by the clients during gaming sessions, the server module is also responsible for coordinating the set up of new gaming sessions, according to the following specifications:

- Upon its activation, the server receives as configuration parameter the number of players that shall take part in the hosted gaming sessions, noted `NUM_PLAYERS`. To simplify automated testing of the project, the initial position of the players in each game is deterministic: all the PacMans shall be placed starting from the top left corner of the screen, vertically aligned one below the other (see next section for more details on the initial placement).
- Next, the server waits till it receives requests to join a gaming session from exactly `NUM_PLAYERS` clients. Only at this point the game is allowed to start.
- For simplicity, the server will host at most one gaming session at a time. Requests for joining a game received while there is already an active gaming session shall be declined or enqueued.

As we will discuss in Section 4, the server module is expected to be replicated across multiple distributed processes in order to mask the occurrence of failures from clients.

3 Client module

The client module shall host, via a Windows form application, the gaming area and the chat.

Gaming area. In order to simplify the development of the gaming part, we have made available a baseline implementation of PacMan in C#[1]. The provided implementation of PacMan runs as a standalone Windows form application (i.e., the whole game logic is hosted on a single machine) and supports only a single player. As such, the provided code will have to be adapted to operate in a multi-player environment and support server-based coordination of the actions submitted by concurrent players. The key idea is include in the client module only the logic for:

[1]The provided baseline implementation is, in its turn, based on a tutorial publicly available at this URL:`http://www.mooict.com/c-tutorial-create-a-simple-pac-man-game-in-visual-studio/`

1. gathering input from keyboard
2. transmitting the player's input for the current round to the server
3. waiting for the corresponding state update from the server and accordingly update the gaming area.

It is responsibility of the server side to determine which players' commands shall be accepted for the current round and the corresponding updates to the game's state.

In this multiplayer variant of PacMan, a game ends when either all PacMans are dead or when all coins have been collected. The winner of the game is the player that collected the largest number of coins. The positions of two players may overlap. The starting position of the Pacman icon for the player who enters the game in the i-th position should be at the following coordinates: $x = 8, y = i * 40$. In order to ensure that, using this formula all the players fit in the screen, the project will be tested with at most 6 PacMans. However, the code should not be optimized based on this limit and should be able to support an arbitrary number of players.

In order to automate the testing of the project, students are further requested to develop a simple variant in which the PacMan's movements are not collected from keyboard but from a trace file. The trace file is a text file that contains a line per round of the game, where each line has the format "*round-id, movement*", *round-id* is a sequential integer starting at 0 and *movement*$\in$\{UP, DOWN, LEFT, RIGHT\}.

Chat. Once a game starts, the players can communicate via a textual chat. The chat is activated by pressing the enter button, which moves the focus to the chat's text box and allows users for typing in the text they wish to send. The text is actually transmitted upon pressing again the enter button.

The chat works by broadcasting each message input by a player to all the other players registered in the game. Note that the chat shall be implemented solely via the client modules in a peer-to-peer fashion, i.e., the server shall not be involved in any chat-related activity (except providing information on the set of players active in a game).

Students are requested to ensure that the chat messages are delivered to clients while ensuring *causal order* guarantees.

4 Fault Tolerance

Given the distributed nature of the **DAD-OGP** platform, it is of paramount importance that the occurrence of failures/network asynchrony is masked away (as much as possible) from clients. Given the real-time nature of the considered application, it is in particular desirable to maximize performance predictability in presence of such abnormal events, e.g., by favouring fault-tolerant solutions that minimize the fail-over latency.

Further, **DAD-OGP** shall include mechanisms aimed at ensuring that a gaming session can continue as long as there is at least one connected client, i.e., the game shall not be aborted or blocked if a subset of clients disconnect or are suspected to have crashed. It is up to the students to define, and justify, the policy to employ in case clients are (falsely) suspected to have crashed.

The students are free to use the replication technique they feel more appropriate to solve the problem at hand. Depending on the selected fault-tolerance strategy, the students shall state how many faulty replicas, f, can be tolerated out of the total number of available ones, replication factor, as well as what are the assumptions on the synchrony model (e.g., synchronous, partially synchronous, asynchronous) of the system. Throughout the project,

students can assume that processes can be subject to crash failures, but do not otherwise deviate from their expected behaviour.

5 PuppetMaster

To simplify project testing, students shall also develop an additional centralised component, called *PuppetMaster*. The role of the PuppetMaster process is to provide a single console from where it is possible to control experiments. The actual specification of the experiment is defined via a, so called, *plot* file: the PuppetMaster reads the plot file and and accordingly steers the distributed computation, by activating clients/servers and orchestrating their execution.

Each physical machine used in the system will also execute a process, called PCS (Process Creation Service), which the PuppetMaster can use to launch client/server processes on remote machines. For simplicity, the activation of the PuppetMasters and of the process creation service will be performed manually. The PCS on each machine should expose a service at an URL on port 11000 for requesting the creation of node replicas on the local machine. This service can be used by the PuppetMaster to create new replicas. For simplicity, we assume that the PuppetMaster knows the URLs of the entire set of process creation services. This information can be provided, for instance, via configuration file or command line.

The first part of each plot file defines the set of processes that should be started by the PuppetMaster via the PCSs. This is achieved via the following syntax:

- `StartClient` `PID` `PCS_URL` `CLIENT_URL` `MSEC_PER_ROUND` `NUM_PLAYERS` [*filename*]: activates a client with a unique identifier `PID` (of type String) via the PCS listening at the url: `PCS_URL`. The client shall expose its services at the address `CLIENT_URL`. If the optional parameter *filename* is specified, the client shall feed its actions from the specified trace file. Else, commands are read from keyboard. The parameters `MSEC_PER_ROUND` and `NUM_PLAYERS` specify the time duration of a round in msecs and the number of players in each game.
- `StartServer` `PID` `PCS_URL` `SERVER_URL` `MSEC_PER_ROUND` `NUM_PLAYERS`: activates a server with a unique identifier `PID` (of type String) via the PCS listening at the url: `PCS_URL`. The server shall expose its services at the address `SERVER_URL`. The parameters `MSEC_PER_ROUND` and `NUM_PLAYERS` specify the time duration of a round in msecs and the number of players in each game.

Additionally the PuppetMaster may also send the follwing commands:

- `GlobalStatus`: This command makes all processes in the system print their current status. The status command should present brief information about the state of the system (who is present, which nodes are presumed failed, etc...). Status information can be printed on each nodes' and does not need to be centralized at the PuppetMaster.
- `Crash` `PID`. This command is used to force a process to crash.
- `Freeze` `PID`. This command is used to simulate a delay in the process. After receiving a freeze, the process continues receiving messages but stops processing them until the PuppetMaster "unfreezes" it. The goal of this command is to simulate network partitions that prevent the successful delivery of message to frozen processes: therefore frozen process cannot explicitly send back any reply (including exceptions).

- `Unfreeze PID`. This command is used to put a process back to normal operation. Pending messages that were received while the process was frozen, should be processed when this command is received.
- `InjectDelay` *src_PID dst_PID*. This command forces the introduction of a communication delay from on the channel connecting *src_PID* to *dst_PID*. This delay should be injected artifiicially at the sender side.
- `LocalState PID` *round_id* . This command obtains a textual representation of the status of the game at a client or server process having identifier `PID` for round *round_id*. The various Pacmans shall be identified with a different integer value (starting from 1), monsters with a M, walls with a W and coins with a C, as illustrated in the example below. The goal is to allow the use of the "diff" tool to compare automatically the local states gathered across different processes. The output of this command shall be visualizable in the PuppetMaster console as well as saved into a file (in the PuppetMaster's filesystem) called `LocalState-PID-`*round_id*. The state of the game should be output as a list of coordinates of all entities including all monsters, all players (indicating whether they are still playing or have lost with the P or L character) and all visible coins, as in the example below:

```
M, 45, 50
M, 120, 60
P1, L, 130, 80
P2, P, 50, 200
C, 40, 40
C, 40, 80
```

The PuppetMaster should have a simple console where a human operator may type commands when running experiments with the system. Also, to further automate testing, the PuppetMaster can also read a sequence of such commands from a *script* file. A script file can have an additional command that controls the behaviour of the PuppetMaster itself:

- `Wait` *x_ms*. This command instructs the pupper master to sleep for x milliseconds before reading and executing the following command in the script file.

For instance, the following sequence in a script file will force broker *broker0* to freeze to $100ms$:

```
Freeze PID
Wait 100
Unfreeze PID
```

All PuppetMaster commands should be executed asynchronously except for the `Wait` command.

6 Final report

Students should prepare a final report describing the developed solution (max. 6 pages). In this report, students should follow the typical approach of a technical paper, first describing the problem they are going to solve, the proposed solutions, and the relative advantages of each solution. However, please avoid including in the report any information included in this project description. The report should include an explanation of the algorithms used and justifications for the design decisions. The project's final report should also include some

qualitative and quantitative evaluation of the implementation. The quantitative evaluation should focus on the following metrics:

- Maximum throughput achievable by the system while varying the number of players and server replicas involved in a gaming session.
- Fault-tolerance features.
- Other optimisations.

This should motivate a brief discussion on the overall quality of the protocols developed. The final reports should be written using LaTeX. A template of the paper format will be provided to the students.

7 Checkpoint and Final Submission

The evaluation process includes an intermediate optional checkpoint and a mandatory final submission. For the checkpoint the students may submit a preliminary implementation of the project; if they do so, they may gain a bonus in the final grade. The goal of the checkpoint is to control the evolution of the implementation effort. Given that students are expected to perform an experimental evaluation of the prototype, it is desirable that they have a working version by the checkpoint time. In contrast to the final evaluation, in the checkpoint only the functionality of the project will be evaluated and not the quality of the solution.

For the checkpoint, students should implement the entire base system, excluding the fault-tolerance mechanisms. After the checkpoint, the students will have time to perform the experimental evaluation and to fix any bugs detected during the checkpoint. The final submission should include the source code (in electronic format) and the associated report (max. 6 pages). The project *must* run in the Lab's PCs for the final demonstration.

8 Relevant Dates

- November 10^{th} - Electronic submission of the checkpoint code;
- November 13^{th} to November 17^{th} - Checkpoint evaluation;
- December 7^{th} - Electronic submission of the final code.
- December 10^{th} - Electronic submission of the final report.

9 Grading

A perfect project without any of the fault-tolerant features will receive 14 points out of 20. The fault-tolerant features are worth 6 additional points for a total of 20 points.

The project grading will depend on a discussion at the end of the semester where all members of the groups must be present and where individual grades will be determined. That grade will depend on, besides the quality of the project, the individual performance in the discussion and the lecturer's evaluation.

The project grade (45% of the course's grade) is the *best* of the following two:

- Final_Project_Grade
- 85% of the Final_Project_Grade + 15% of Checkpoint_Grade

10 Cooperation among Groups

Students must not, *in any case*, see the code of other groups or provide their code to other groups. If copies of code are detected, both groups will fail the course.

11 "Época especial"

Students being evaluated on "Época especial" will be required to do a different project and an exam. The project will be announced on January 27th, 2017, must be delivered February 2nd, and will be discussed on February 3rd, 2017.