Ge.neric E.rlang Se.rver

Operating systems and multicore programming (1DT089)

Project proposal for group 10: Johan Gille (930922-3452), Niklas Hökenström (880615-6215), Valle Magnusson (900215-5217), Jonas Nilson (920927-2179), Christian Törnqvist (920104-0616)

Version 2, 2014-05-07

Table of contents

1. Introduction

A large portion of the games available today have some kind of multiplayer support that enables players from around the world to both compete and cooperate in the same game. With this project we are aiming to get a better understanding of the concept of how a game server works and how to implement one of our own. We think, since it is new to all of us, that this would give us some great programming experiences. To test our server implementation we have planned to make a simple game inspired by *Liero* where a large number of players would be able to connect and play in real-time. Even though the main focus will be on making the back end server work the thought of making an online game is very intriguing, so our motivation will be on top.

In this project, we will be able to cover a lot of new ground that we have not tried yet. From basic network programming to game GUI (Graphical User Interface). Our plan is to program the back end server in Erlang while the front end will be in Java. This will require us to learn how to integrate two languages with each other.

The main challenge of this project will probably be designing the server. Since we do not have any prior experience with server programming, we will need to do a lot of reading about what kind of modules we might need and how they should integrate with each other.

One of the other propositions that we discussed was some sort of a Mandelbrot set zoomer. Each pixel of the Mandelbrot set is generated by calculating a simple recursive formula and by determine whether the result diverges after N iterations or not. If it diverges, it is not a part of the Mandelbrot set. Depending on after how many iterations the recursion diverges (diverges in this case is when you reach a number that is greater than 200), the pixel will have a specific color. Also, the color will be black if the pixel is a part of the Mandelbrot set. This will generate incredibly beautiful patterns that can be zoomed infinitely since it is fractals. The work to be made for calculating this would be optimal for multiple cores by simply dividing the image into N different sections and allowing N cores work on these. We figured that this might have been a bit too simple since you can write one of those programs on your own if you give it one day or two.

Another proposition was also regarding fractals in some other way. We thought of some kind of a fractal generator where you could define a pattern and the program would thereafter develop these patterns recursively in N iterations. However, as with the Mandelbrot set zooming program, this was not something that we felt was exciting enough to select as our project.

2. System architecture

This UML (Unified Modeling Language) scheme (figure 1) describes a high level abstraction of our software design.

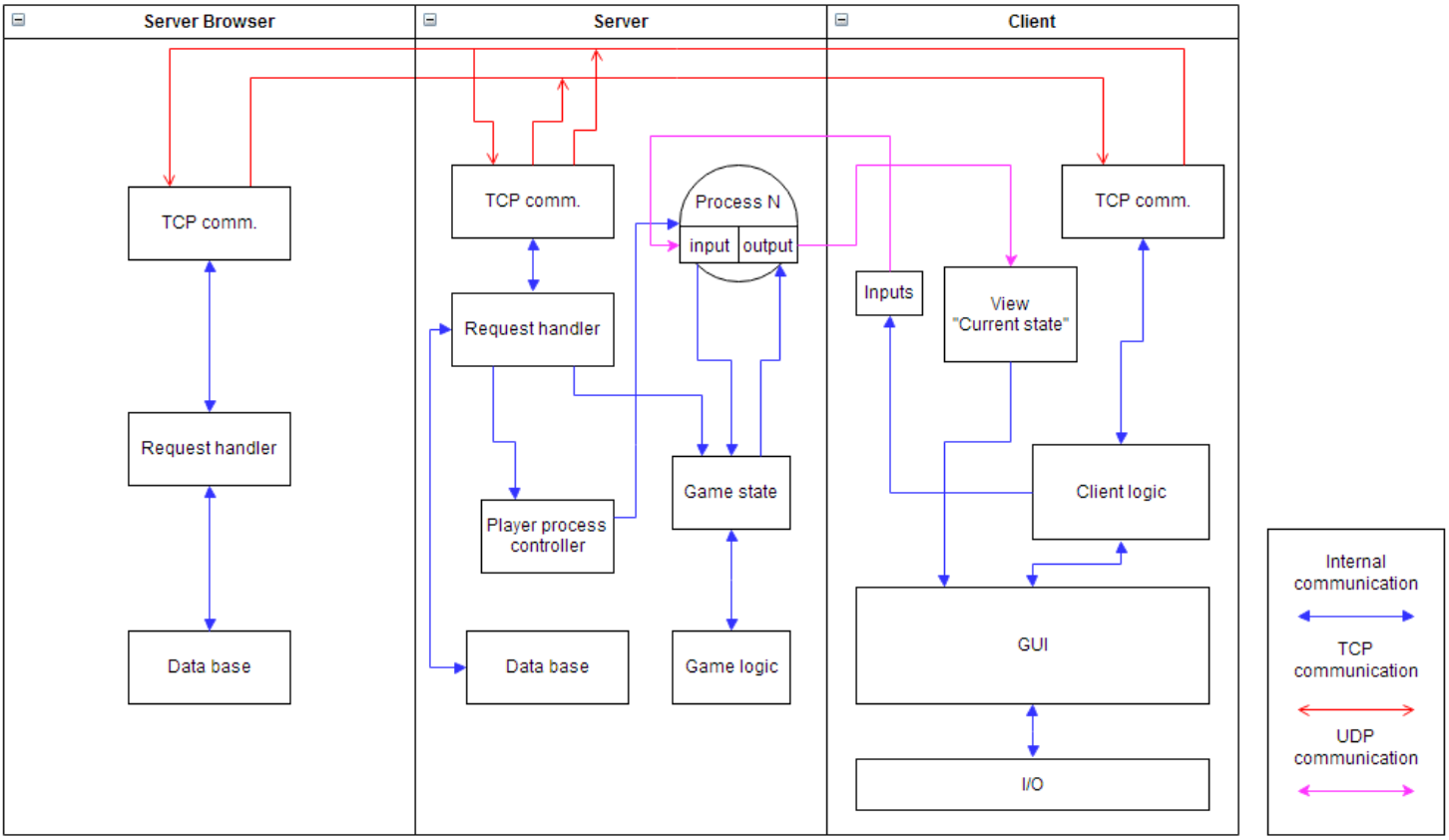

Figure 1 - UML scheme

The system is split into 3 major parts.

- Server

  The server will hold the state of the game and its logic. It will be ready for clients that want to connect through TCP (Transmission Control Protocol) and add them to the game by spawning a new process as well as establishing a UDP (User Datagram Protocol) connection. To make it easy for clients to connect the server will send information, like its address, to the server browser.

- Client

  We want the client to be able to connect a server directly by address but also see available servers. Instead of searching for all servers, the client will just check with the browser for available ones. The client will have a java GUI for the game that will draw the current game state it received from the server. Some predictions logic might also be added here to increase the responsiveness of the game.

- Server browser

  Servers will send information to the browser of their address, name and such. By doing this clients can easily choose a server to connect to. Other features like creating a server could also be added here.

We have an intention of making our server as generic as possible. Thereby, there will be a need for a server API (Application Programming Interface). Since we primarily want to focus on real-time games, there will be need for server functions such as for instance: loadWorld(), drawWorld(), spawnCharacter(), moveCharacter(), fire(), aim(), modifyHP(), modifyLifeCount(), modifyScore(), et cetera. If we want to add turn-based game functionality, we will have to add some additional specific functions. Some of these may be lockPlayer(), unlockPlayer() and so on.

For the game which will be written in Java, we are going to implement the model-view-control design pattern. We have not yet decided which parts that should be client or server-side. However, since we want a fluent and responsive game, we will probably have the client's own character movement occur on the client's side and meanwhile, the client's character position will update on the server model. This means that we will have one model-module on the client side and one model-module on the server side.

The internet protocol suite that we intend to use is UDP. This is because we want to have as low latency as possible which is required by real-time games. Packet loss from UDP would not be a problem because only the most recent information would be relevant. We are also looking into complimenting our connection with TCP. We would use TCP for communication where packet loss is not ideal for example chat messages and score count.

3. Concurrency models

We will divide our program into two parts, back end and front end. The server, back end, will be implemented in Erlang with the use of the OTP (Open Telecom Platform) framework since this seems to be optimal for server programming. Here we will use the actor model. The specific details of how we are going to use the actor model is not yet entirely explored, but we are thinking about using for instance one process for each client. This way, all clients will be able to communicate with the server concurrently.

The front end will be written in Java. Since we will probably want the game to be web-based, we will use some sort of framework for this. We found a framework called Struts that we will look further into. Currently, we do not have any specific plans for implementing any concurrency at all in our Java program. One thing that might be done concurrently is to update the model while concurrently sending data to the server.

4. Development tools

- Erlang
    - For Erlang development, we will use Emacs for text editing. Some plug-ins to Emacs that might be worth considering is “flycheck”, which dynamically checks for syntax errors etc.
    - For testing we will use EUnit.
    - For documentation we will use EDoc.
- Java
    - For Java development, we have considered Eclipse since it is great for refactoring. We also think that object-oriented programming and IDE’s go hand in hand. There is of course nothing that stops anyone from using Emacs as editor for our Java development if he prefers that.
    - For testing we will use JUnit.
    - For documentation we will use JavaDoc.
- We plan to use Git for version control since it is the one software that we are used to. Our code will be hosted on one of our private github repositories.
- The build tool we intend to use is make. In the beginning of the project, we will create a make file which should be as generic as possible.

5. Process evaluation

Our project proposal process did not contain a full fledged brainstorming process as we were proposed to do. This was due to the fact that we missed the information about the structured brainstorming. We did however have several meetings where we discussed different kind of projects that might need concurrency. From the beginning, we did not feel that we wanted to “force” concurrency into a program that might not actually benefit from it. Therefore, we primarily focused on problems that would mostly rely on concurrency for improved performance. Some of our first ideas were thereby regarding different kinds of fractal generators whose work can be divided into an arbitrary amount of processes. These kinds of projects seemed at glance to be relatively easy to implement and we wanted something more challenging for educational purposes. We all felt that we wanted to do something with Erlang, and since Erlang is optimal for networking, we thought that we should do some sort of a server. The fact that no one of us has ever done anything with networking before and since this is a major part of computer science, this seemed like a good idea. Simultaneously, we wanted some kind of end result that you can actually be entertained with, so we thought of a game

server and also, to create a simple online game. We wanted a language that was simple to write games in so we chose Java for this part. Since the original intent is to have the client play the games through an internet browser, Java seemed like good choice.