ΠΛΗ-414 Κατανεμημένα Συστήματα
Distributed Systems Semester Project 2020

In groups of 2 people

Due date: Friday, May 29 2020

Contents

1 Overview

You will develop a distributed game engine for board games, as a distributed system. The platform will support user authentication, access authorization, multiple games and score boards, tournaments and spectator ability. Two games, tic-tac-toe and chess, will be supported.

1.1 Requirements

1. The system will authenticate all **users**.
2. Each user must authenticate with a password, at an Authentucation Service, which will issue a user token once. Sucbsequently all other services in the system will use this token. New users shall be created by the Authentication Service and will have a unique user name.
3. For each user, the GameMaster Service will be keeping the scores of all **plays**[1] played by this user.

[1] A *play* is a particular round of two players playing a game. A play ends up with either a winner and a loser, or with a draw. Also each player obtains a score

4. The GameMaster Service will be able to support multiple **games**, although you will only implement two: tic-tac-toe and chess. But it should be extensible to more games (e.g., backgammon, othelo, Go, checkers etc).

5. Users will be able to play two types of plays. A **practice play** can be played between two users, which are paired by the GameMaster service arbitrarily.

   A **tournament** is held between a set of players, and it consists of **play rounds** of elimination until the top 4 winners are selected.

   The score of a practice play is recorded seperately from the score obtained in tournament plays. The number of plays each user participated in, the number of wins, ties and losses, and the total score of each play are kept.

   That is, for each user there are three scores:

   Practice scores: The scores of individual plays are available only to the players of these plays (a player can see the full list of practice scores she played). A player can only see the total score of other players.

   Tournament scores: The scores of each individual play are available to all players.

6. User roles are the following:

   Player: These users will be able to practice and participate in tournaments.

   Official: These users will be able to create tournaments.

   Admin: These users will be able to perform administrative operations.

2 Implementation

Each project team will consist of 2 members, and will implement their own set of services, following the micro-service-based 3-layer architecture.

An overview of all services can be seen in Fig. 1. The front end services are as follows:

User Interface Service: This service is responsible for providing a *web-based* user interface to the application logic. The web-based user interface does not need to be fancy; you can use whatever technology you like (simple form-based, or Ajax).

Web browser: It is up to you to decide what part of the logic (if any) will be executed in the browser via javascript, and what will be left to the web server.

A brief description of the middle tier services is as follows:

Authentication Service: it is responsible for maintaining the user authentication information and other relevant personal info (e.g., user email), as well as the roles that each user has.

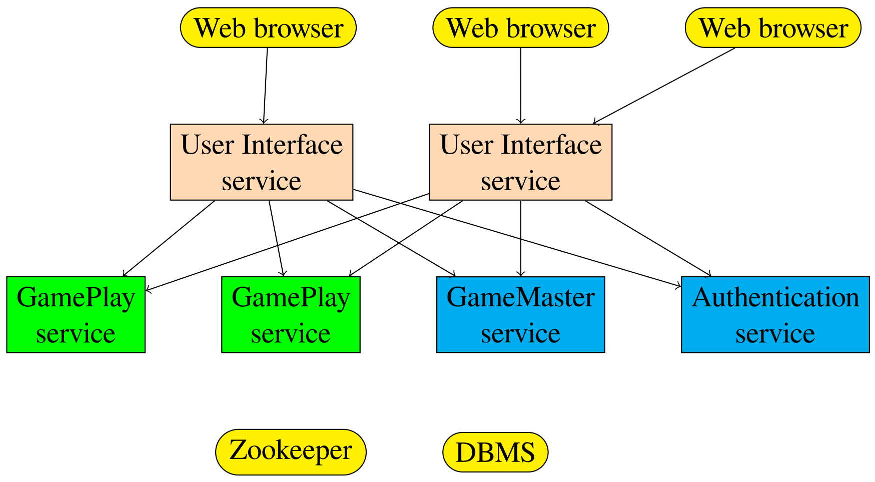

Figure 1: An overview of the services participating in the platform.

.

A user can self-register to this service and is assigned the role of Player. An Admin user can assign additiional roles (Official and/or Admin) to users.

GameMaster Service: This service is responsible for keeping the player scores, for pairing players for practice plays, and for creating and managing tournaments.

PlayMaster Service: This service is responsible for managing individual plays. When a play is created (either a practice play or as part of a tournament), it is assigned to some PlayMaster Service.

There are several playmaster services. Each PlayMaster service can manage several plays concurrently. If a playmaster service fails, the games it manages are assigned to a new PlayMaster service.

Finally, the back-end services should include the following (at a minimum):

Zookeeper: You should use Apache ZooKeeper (`https://zookeeper.apache.org/`) for configuration, service directory and initialization. In addition, you can use zookeeper to detect failed PlayMaster services and reassign Plays to new PlayMasters.

DBMS: You can use one or more databases for the application. At least one database is necessary. You may use either some SQL database or a NoSQL database, or both types.

A suggestion is to use an SQL database for authentication and GameMaster, and a NoSQL database with replication for PlayMaster services.

2.1 Non-functional requirements

1. Your system must be implemented so that each microservice instance will be on a different "node". To this end, you will use *containers*. It is suggested to use **docker** (`https://www.docker.com/`), since it is—by far—the easiest to use.

2. If a PlayMaster service node fails, the system must be able to continue with minimal interruption.

3. Implementation language: you are free to use any language and set of libraries you want. Keep in mind that you can implement different services in different languages (for example, javascript+nodejs for the UI, Python or Java for the rest).

   Note that depending on the language/framework you use, I may be able to help you more. Note that I am not familiar with PHP and Ruby/rails (among the popular ones). Instead, I can help you with Java, and especially with Python.

   Note that this should not discourage you from using something more "exotic", such as Go, Ruby, Scala, or Groovy, on the contrary, I will be glad if you do...

3 Deliverables

You should be prepared to deliver:

1. Full source code of all parts of the system (including Makefiles, Dockerfiles, Ant/Maven files etc). **Just delivering the project directory of some IDE such as Eclipse, Netbeans or IDEA, is not sufficient.**

   Ideally, one should be able to install your system on docker by running a simple script (but you may need to provide some assumptions on the docker setup you chose, especially if you do not use Docker-Swarm.

2. A presentation (in Openoffice, Powerpoint or PDF) describing your implementation, including

   (a) Tools, frameworks and libraries you used

   (b) Choices that you made (e.g., how did you implement the web interface)

   (c) Execution on 8 nodes or more (see the *Okeanos services* `http://courses.ece.tuc.gr-`Χρήσιμα) and measure the performance of some indicative service calls under concurrency (use Apache Bench for this)

   (d) System recovery response, if a PlayMaster node fails.

3. You should be prepared to present the above presentation on the last week of classes at the end of the semester.

4 Advice

To properly implement your services and their functionality, you should use libraries and frameworks for providing major services. If you have already used such software, it is acceptable to use them for this project. However, a goal of this project for you should be to familiarize yourselves with new software, so do not hesitate to try learning new things.

To implement some parts of the whole system, you will need knowledge that we will study later in the course. So that you make continuous progress on the project, it is important to plan your coding and studying work, on those parts that are covered first.

Here is a short list of advice.

- First, form a team (2 people per team) with someone, exchange emails and other contact information. Agree on the tools to work with each other. This is extra important since you are going to be working without meeting each other.
  - A teleconferencing platform. Zoom is a good choice in this regard.
  - A platform where you can message each other quickly (email can be hard to use sometimes). I use `slack` and can recommend it.
  - A platform to exchange source code. If you are not on `github`, get on it now! Also, make sure you know how to use *git*
- Start selecting the software that you will use to implement your project. I strongly recommend one of the following "language ecosystem" choices:
  - Java ecosystem (including scala or groovy if you know or want to learn them) is of course an excellent choice. This is the preferred ecosystem for information systems around the world, and learning it better is a task that you will not regret.
  - Python is also an excellent choice (my personal favorite), and offers many excellent facilities for implementing our project. Python has recently become the #1 language for data analytics, machine learning and high-performance computing. You can't go wrong with it.
  - Javascript ecosystem, including `nodejs` and `npm` is also a good choice. Since some javascript will be needed at the level of implementing the web-based, you might want to go all the way and learn how to write back-end code in JS as well. If you want to become serious for front-end development, a good knowledge of JavaScript is a must.

  Once you have selected the ecosystem, it is a good idea to also select a good build system for your code and get familiar with it.
- Start playing with `docker` (highly recommended) or any other platform for deploying containers (you are on your own!!). In particular, learn how to build

a Docker image and how to deploy a swarm. There is excellent documentation on the Docker website.

- The next step is to design your implementation. Make sure you are fully aware of what the service API of each of the middle tier services is. Use UML a lot to specify your design, and then code your UML. If not, you will forget what your design is, and it will be hard to retrieve it from source code.

- Make sure that you write unit tests for every piece of code that you use. Unit testing is a very useful habit that you need to acquire as programmers, and it is going to save you 50% of the effort you will otherwise put in building your software.

  A very important use of unit testing is to test tools that you test the libraries and tools that you will use. Write small tests to use the tools the way you intend to use them in the code. These tests will make you more confident that you have understood the use of these tools, and are a great way to learn new APIs and libraries.

- Share the work. One of you should take charge for each new technology that you learn. For example, one can be in charge of docker and deployment, another can be in charge of web development. Both of you should probably work on the middle tier services, on Zookeeper and the databases. What one learns, she transfers to the other.

Good luck, and remember:

KISS : Keep It Simple Stupid!
and
RTFM : Read The Fine Manual!