

Fail-Safe Cloud Tournament Engine with Error Detection and Error Recovery



UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

Kyle Chapman (20703236)
Supervisor: Dr. Cornelia P. Inggs
Cosupervisor: Mr. Andrew J. Collett

9 November 2020

Contents

1	Introduction	2
1.1	Scope	3
1.2	Document Outline	3
2	Overview	4
2.1	Separate enviroments	4
2.2	Web User Interface	5
2.3	Software Stack	5
3	Design and Implementation	7
3.1	Overview	8
3.2	Web User Interface	8
3.3	REST API	8
3.4	Database	8
3.5	Database Watcher	8
3.5.1	Match Server	9
3.5.2	Match Client	9
4	Testing	10
5	Future Work	11
5.1	Kubernetes	11
5.2	Other Game Types	11
6	Conclusion	12
7	Appendix	14
7.1	Web User Interface	14

Chapter 1

Introduction

The Fail-Safe Cloud Tournament Engine (FSCTE) is an extension of the Cloud Tournament Engine (CTE) developed by Reece Murray in 2018, which is an extension of the Tournament Engine (TE) built on the Ingenious Framework¹ (IF). The goal for the CTE is to allow people to play a turn-based game, provided it is supported in the IF, against each other in a match, where many matches make up a tournament. Multiple tournaments can be run to determine how good each player is. Tournaments can be separated into two categories: private and public.

Private tournaments are tournaments that only the administrators can add players to, and they are mainly used to test some players against other players. Public tournaments are tournaments that any user can add their own players to and administrators can also add players to. These tournaments are used to put players against each other to rank each player based on how many wins they get against other players.

It was found that there were some major limitations to the CTE, such as minimal error reporting, lack of error recovery and an unmaintainable code base to name a few. As a result, there were situations that arose in which a player failure resulted in a failure of the entire tournament. This prompted the need for a version that was fail-safe and included error reporting and error recovery, as far as possible. The FSCTE aims to address these limitations and improve the overall system stability by being fail-safe, which means that if any error were to occur, the response to the error would cause as little harm to the overall system stability as possible.

¹<https://bitbucket.org/skroon/ingenious-framework/src/master/>

1.1 Scope

The system used to manage the FSCTE is made up of multiple Docker² containers, which run each section of the FSCTE in their own separate environment. If some error occurs in a section of the system, it will not affect the other sections, since they are in separate environments.

The FSCTE uses the following three main sections that are run in their own Docker containers:

- **Web User Interface:** Allows a user to interact directly with the FSCTE.
- **Database:** Stores all the information relevant to the FSCTE.
- **Database watcher (Golang):** Watches the database for any changes, such as tournaments being started, and sends messages to the front-end accordingly.
- **REST API:** Facilitates file upload to, and download from, the web user interface, as well as CAS authentication for logging in with Stellenbosch University credentials.

The layout of the entire FSCTE system is further discussed in chapter 3, *design and implementation*.

1.2 Document Outline

This report will discuss the overview of the system in chapter 2, the implementation and design of the system in chapter 3, then onto the testing in chapter 4 and discussion of the future of the project in chapter 5. Finally, the conclusion will be presented in chapter 6.

²<https://docs.docker.com/engine/docker-overview/>

Chapter 2

Overview

This chapter will give an overview of the requirements for the FSCTE system and how the requirements have been met.

The CTE used in 3rd year at Stellenbosch University is designed to manage many Othello [1] tournaments, involving many players, concurrently. A user is able to view public tournaments, players, referees, schedulers, etc. and view relevant statistics for ongoing and completed tournaments. The admin, usually the lecturer, is able to add tournaments, schedulers and rankers and the students are only allowed to upload their players and add them to public tournaments.

2.1 Separate environments

Each section of the FSCTE needs to be in their own separate environment, so that if some error occurs, it impacts only that specific section and not the entire FSCTE. Docker was chosen for this purpose, since it is relatively easy to spin up a section in its own environment and redeploy that section if some error occurs.

The docker container is created by writing a **Dockerfile** that specifies which base environment you want to use (e.g. **ubuntu**, **node**, **neo4j** etc.) and the command you want to run in the container. The container exits when the command finishes, so in order to keep the container running forever, infinite loops are useful. This means that the command will only finish when the user forcefully stops the container.

Docker-compose¹ is used to spin up all of the docker containers used by the FSCTE, using their respective **Dockerfiles**, and allows for more control of how the container should be set up. Local directories can be mounted into the docker containers so that any files that are generated inside the container can be accessed

¹<https://docs.docker.com/compose/>

outside of the container, on the host machine. This is useful for accessing the log files after a match has completed, so that it can be seen if the match was a success or not.

2.2 Web User Interface

The previous web user interface implemented in the CTE was not very user-friendly, which resulted in a lot of confusion for the students who used the CTE. The entire web user interface in the FSCTE was written from scratch using new technologies, such as Vue² and TypeScript³, so that it was more visually appealing and easier for the lecturers and students to use. Students and lecturers (or admins) can log into the web user interface using Stellenbosch University Single-Sign On or by specifying their username and password if they already exist in the database.

The students are able to upload their players to the web user interface and join public tournaments with their players. The students can have as many players as they would like, and each of the players are added to the tournament standings once they join a tournament. This allows the students to see how their players perform relative to other players in the same tournaments.

The lecturers, or admins, are able to upload their own rankers, schedulers, referees and players to the web user interface. More will be explained about the rankers, schedulers and referees in chapter 3. The admins can also add their own players to public or private tournaments, as well as they can add a student's player to a public or private tournament. The admins can start and stop tournaments, as well as create new tournaments.

After a match has completed, either ending in a failure or success, the log files of that match can be accessed if the student or admin desires. This means that either the student or admin can see what moves both the players made and the final result of the match.

2.3 Software Stack

The FSCTE uses the following software stack:

- **Vue** - Front-end framework that is used to interact with the FSCTE.

²<https://vuejs.org/>

³<https://www.typescriptlang.org/>

- **Typescript** - Superset of JavaScript that adds types, which is useful for catching bugs before they make it into production.
- **Python** - Used to create the REST API that facilitates file upload to, and download from, the web user interface, as well as providing authentication for logging in using Stellenbosch University Single-Sign On.
- **Neo4j** - Database that is used to store all the information necessary for the FSCTE to work. Highly relational database that uses the Cypher⁴ query language to get or set attributes.
- **Golang** - Used to schedule matches in tournaments concurrently, with less overhead than using Java.
- **Docker** - Tool to allow different components of the FSCTE to operate in their own separate environments.
- **Bash** - Used for writing scripts that will setup the FSCTE and download any necessary programs.

The host system does not need to have docker pre-installed, since the FSCTE will check if docker and docker-compose are available in the setup script. If one or more programs are not installed, the script will download the necessary version onto the host machine.

⁴<https://neo4j.com/docs/cypher-manual/current/>

Chapter 3

Design and Implementation

This chapter will describe the design and implementation of all the components present in the FSCTE. Figure 3.1 shows an overview of the components present in the FSCTE and how they interact.

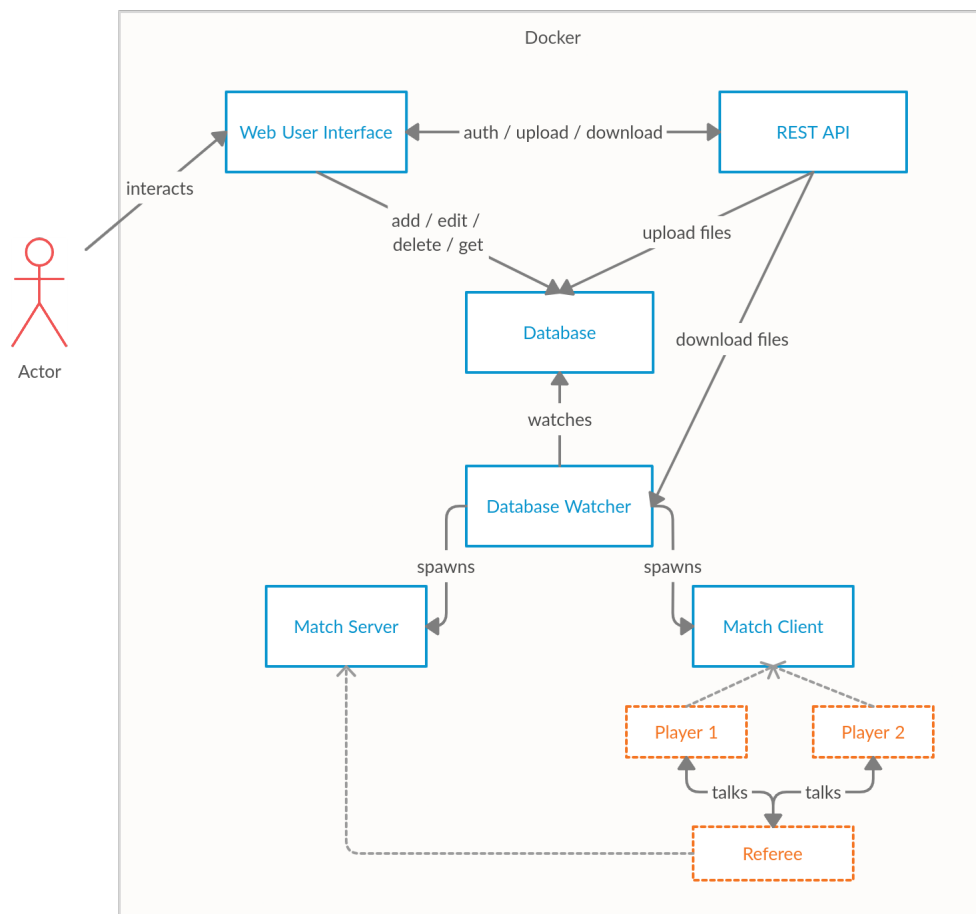


Figure 3.1: UML diagram representing the structure of the FSCTE

3.1 Overview

This section serves as an explanation for what can be seen in the UML diagram in Figure 3.1.

The actor represents a user, who can be either an admin or student. This user only interacts with the web user interface to add players, tournaments and anything the user desires, provided they have the correct permissions.

The rectangles with a blue outline represent docker containers and the rectangles with a dotted orange line represent entities inside the docker containers. The dotted grey line represents the docker container in which an entity resides.

3.2 Web User Interface

This section serves as an explanation for the purpose of the web user interface docker container seen in Figure 3.1.

3.3 REST API

This section serves as an explanation for the purpose of the REST API docker container seen in Figure 3.1.

3.4 Database

This section serves as an explanation for the purpose of the database docker container seen in Figure 3.1.

3.5 Database Watcher

This section serves as an explanation for the purpose of the database watcher docker container seen in Figure 3.1.

3.5.1 Match Server

3.5.2 Match Client

A scheduler is used to schedule matches between two players in a round-robin fashion, where every player plays against every other player. After every player has played against every other player, the overall win-loss ratio can be calculated that can be used to distinguish between good, average and bad players. The win-loss ratio and the Elo rating [2] can be used in conjunction.

A ranker is used to distinguish between the good and bad players by comparing the Elo of the two players, where every player starts on the same amount of Elo. For every win that a player has, they gain a certain amount of Elo and for every loss that a player has, they lost a certain amount of Elo. The amount gained or lost decreases for every match that the player plays, meaning that eventually the Elo rating will be a true reflection of the skill level of the player.

A referee is used to monitor the moves that the two players make in a match and check if any player makes an invalid move or times out. One player sends a move to the referee, which then checks if the move is valid and only sends the move to the other player if the move made is valid. If the move made is invalid, or the player times out, the match will be forfeit and the last player to send a valid move becomes the winner.

Chapter 4

Testing

Chapter 5

Future Work

This chapter covers the future work that could be implemented in the Fail-Safe Cloud Tournament Engine (FSCTE) to extend the current functionality. The conclusion will be provided in section 6.

5.1 Kubernetes

Kubernetes¹ can be used to manage the docker containers, so that if a container crashes, another instance of that container can be redeployed without hesitation. Due to time constraints, it was not feasible to add Kubernetes to the current FSCTE.

5.2 Other Game Types

In the current FSCTE implementation, only the `Othello` game is supported. Supporting other game types will allow the FSCTE to be more abstract and versatile. Due to time constraints, it was not feasible to attempt to support other types of games.

¹<https://kubernetes.io/>

Chapter 6

Conclusion

The main requirements of the FSCTE that needed to be satisfied were:

- have effective error handling
- have effective debugging facilities
- scale with tournaments and players
- have a logical, clean and maintainable code base
- have a more user-friendly web user interface

All of the above requirements were satisfied. The error handling requirement was satisfied by making sure that at any point where an error could occur, it was decided if the error was serious or not. If the error was serious, then an error message would be displayed and the component that caused the error would be stopped. If the error was not serious, then an error message would be displayed, but nothing would happen to the component that caused the error.

The effective debugging facilities requirement was satisfied by logging all information that happens inside each docker container, so that if some error occurs, the user can look in the docker logs to see at which stage the error occurred. This allows the user to know exactly what was going on before the error occurred so that the user can pinpoint the issue and fix the problem.

The tournaments and player scaling requirement was satisfied by writing the code base in such a way that it performs very well for any number of tournaments and players. Golang was mainly used to allow scaling to be possible through the use of concurrent threads running different tasks at the same time.

The maintainable code base requirement was satisfied by providing extensive documentation throughout the code base, so that anyone who reads the code base

can understand what is going on without any issue.

The user-friendly web user interface requirement was satisfied by ensuring that the user can navigate to any page or get any information they want in a maximum of two mouse clicks. A lot of time was spent making sure that the web user interface looked clean and visually appealing, as well as easy to use.

Working on the FSCTE for the past year has been a great challenge, but also a great reward. Initially, it was tough to understand what was going on in the CTE because of the lack of documentation and confusing way that some sections were implemented. After understanding the previous work, the toughest challenge was getting the players to play a match and communicate their results to the web user interface. However, it was a fun and informative experience.

Chapter 7

Appendix

7.1 Web User Interface

The images from Figure 7.1 to 7.5 are screenshots from the web user interface that the admins and students will interact with.

RW314 Tournament Engine [TOURNAMENTS](#) [MATCHES](#) [STANDINGS](#) [ABOUT](#) [LOGIN](#)

Login

Username and password

LOGIN

Stellenbosch University Login


SU LOGIN


Copyright © 2020 - Kyle Chapman

Figure 7.1: Login page

Dashboard

Othello Tournaments

Public Tournaments		Search Tournaments 	
Tournament Name ↑	Status	Turn Timeout (ms)	Players
Placements	Not Started	5400	0
Rows per page: 10 1-1 of 1 < >			

Private Tournaments		Search Tournaments 	
Tournament Name ↑	Status	Turn Timeout (ms)	Players
Gold	Not Started	4000	0
Platinum	Not Started	4000	0
Silver	Not Started	4000	0
Rows per page: 10 1-3 of 3 < >			

[CREATE TOURNAMENT](#)

Matches

No matches have been created yet.

Figure 7.2: Dashboard after logging in as admin

[< BACK](#)

Tournament Placements

[COLLAPSE ALL](#)

Tournament type
^

Public

Join tournament
^

Add a player:

ADD PLAYER

Select student player to add:

ADD

Status
^

Not Started

Manage the status

START

Time limit per move
^

5400 milliseconds

Referee
^

OthelloReferee

Scheduler
^

RoundRobin

Ranker
^

EloRanker

[EDIT](#)

[DELETE](#)

Players Currently In Tournament

Players		Search Players		
User	Player Name ↑	Date Added	Remove Player	
No data available				
		Rows per page:	10	- < >

Figure 7.3: Page describing the tournament ‘Placements’

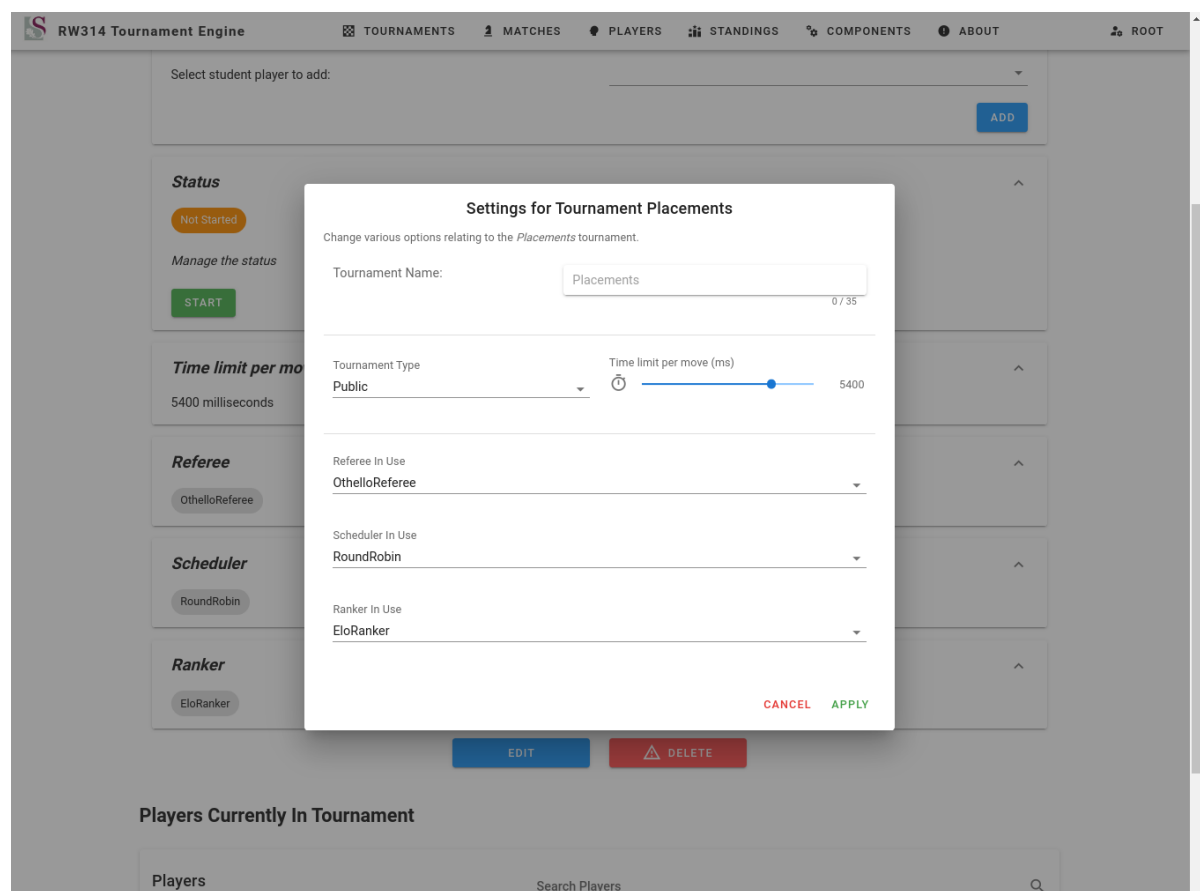


Figure 7.4: Settings popup for the tournament ‘Placements’ when the logged in user is an admin

About

RW314 Tournament Engine was designed to allow students of the 3rd year class *Concurrency* to test their Othello players against each other in various tournaments.

Types of tournaments

- Public** Any player can join the tournament.
- Private** Only the admins can add players to the tournament.

Tournament information

Each tournament is made up of *three* components, which control how the tournament is run, how winners are decided and how the standings are managed.

- Ranker** Decides how many points to award the winner of a match, and how many points to take away from the loser. If a match ends in a draw, no points are awarded to either player. There are varying ways of measuring the number of points to award and take away, such as the [Elo rating system](#).
- Referee** Manages a match between two players, monitoring the moves made by each player to ensure that each player makes a valid move. If a player in a match makes an invalid move, or throws a **segmentation fault**, the other player automatically wins the match and the rest of the match is not played out.
- Scheduler** Decides how the matches between two players are assigned, such that ultimately all players will play all other players. There are various ways of assigning the matches, such as in a [Round-robin](#) or [First-Come First-Serve](#) fashion. Eventually, the *ranker* will be used to schedule matches between players of roughly equal skill.

Figure 7.5: About page

Bibliography

- [1] Masters Traditional Games. *Rules and Instructions for Reversi and Othello*. 2019. URL: <https://www.mastersofgames.com/rules/reversi-othello-rules.htm>.
- [2] Adam Newell. *What is Elo? An explanation of competitive gaming's hidden rating system*. Jan. 2018. URL: <https://dotesports.com/general/news/elo-ratings-explained-20565>.