Cloud Tournament Engine

Reece Murray, 18322891

November 5, 2018

Contents

Chapter 1

Introduction

This report covers the design and implementation of the Cloud Tournament Engine (CTE). The CTE is built on the implementation of a similar project, called the Tournament Engine (TE). The TE facilitates the creation, pausing, resumption and termination of tournaments that have been built on the underlying Ingenious Framework (IF) [10]. The CTE is an extension of this project that allows the TE to spawn servers, databases, referees, or players on different physical machines, whether it be on a cluster within the university, or computing on cloud infrastructure. This functionality allows for the user to specify the compute resources[1] to allocate for a specific tournament.

1.1 Scope

The user interface component of the CTE is hosted on a web server. This interface allows the user to create, manage and monitor tournaments. The project back-end consists of four major components, each built to run independently in a Docker container [3]:

Web Server hosts the web interface for the user to interact with the CTE.

Database Server hosts the database for the CTE, multiple web servers can utilize a single database.

Tournament Referee container executes the IF referee to interact with tournament clients.

Tournament Client runs a tournament player on a remote computer.

The system orchestrates the distributed computing of the CTE by utilizing Docker swarm [9]. Docker swarm allows for computers to be added to a swarm as nodes in a cluster. This provides the CTE with automatic load balancing of compute resources for applications launched to run on the swarm, allowing for the resource constraints (specified by the user) of a running tournament to be upheld. Figure 3.1 shows the communication and layout between the above mentioned components. More about this will be discussed in Section 3.

1.2 Document Outline

This document will provide an overview of the high-level requirements of this project, as well as an overview of the different components that make up the CTE. This will follow into the high-level design and implementation of the CTE. Each component will be broken down into sub-components and explained. Testing will be covered on a component and integration testing basis. Lastly, future work will be proposed, followed by a conclusion.

[1]The amount of RAM (memory) and CPU

Chapter 2

System Overview

This chapter will provide an overview of the high-level requirements of this project and how the CTE satisfies these requirements. The sections that will be covered are the cloud, user interface, hardware requirements, software requirements, functional requirements, and deployment.

2.1 The Cloud

A requirement for the Cloud Tournament Engine is the ability to host the system across multiple machines (potentially machines in the cloud) and utilize the resources of each machine. The CTE makes use of Docker to satisfy this requirement. Docker provides a range of features that help implement the distributed computing functionality of this project. Some of these features include the containerization of components, Docker network [5], which allows communication between different containers, and Docker swarm, which provides the functionality to distribute tasks to computers in the cloud. More about this will be mentioned in Chapter 3.1.

2.2 User Interface

This section will cover the user interface, its purpose, and which elements were changed in the original TE to accommodate the CTE. The TE user interface allows the user to create, join, manage, or terminate a tournament. Users can upload their game schedulers, rankers, players and tournament configuration files. These files can then be used to start a tournament. It was necessary to change a few key aspects in the TE's user interface to reach the goals of the CTE. These changes included a live updating system, user specified constraints when creating a tournament, and support for the Othello Tournament for the RW314 course at Stellenbosch University. More about these changes will be mentioned in Section 3.

2.3 Hardware Requirements

The system is built to run in the cloud. The minimal hardware requirement is a single server for the CTE to be executed on. However, the resource requirements of the players involved in tournaments is satisfied by how many computers (their available resources need to be considered) are available in a swarm for the CTE to utilize.

2.4 Software Requirements

The system makes use of a LAMP stack for handling user interaction. The entire system makes use of PHP, JavaScript, Java, HTML, Bash, and Golang. All system components run within Docker. It is required that the computer hosting the system has Docker-CE installed [4]. Each computer that hosts tournament players needs to be connected to Docker swarm as a node and use the same Docker version. The Docker version needs to be 18.06.1-CE or higher.

2.5 Functional Requirements

This section will introduce the initial project requirements as per the project brief.

2.5.1 Selection of resources for tournaments

The CTE should have the functionality for the user to allocate the required resources (*i.e.* the amount of memory or CPU utilization) for the player to use when computing moves. This functionality has been implemented using Docker's built in resource constraints [7].

2.5.2 Spawning of new resource nodes

The CTE should support the ability to add more cloud servers when more resources are required by players. The web server will allow the user to select an available node from the web interface. When a new node is added to the cluster, it should automatically be added as an available resource within the database. This functionality has been implemented using the Docker swarm. The Docker swarm also manages load balancing, and thus, a user does not need to specify where a player is hosted.

2.5.3 Multiple concurrent tournaments

The CTE should support multiple tournaments running concurrently. This functionality has been implemented using an on-demand programming approach. More about this will be mentioned in Section 3.2.2.

2.6 Deployment

The CTE can be deployed on a single machine, or across multiple machines on a swarm in the cloud, see Appendix A for more information about the swarm and deploying the CTE.

Chapter 3

Design and Implementation

This chapter will sketch the high-level design of this project. The design of this project consists of five major components. The following sections will cover why these components were built this way, and how these components interact to form the CTE. Figure 3.1 displays a broad layout of the components and how they communicate. This chapter will define the specific details about each component and its sub-components.

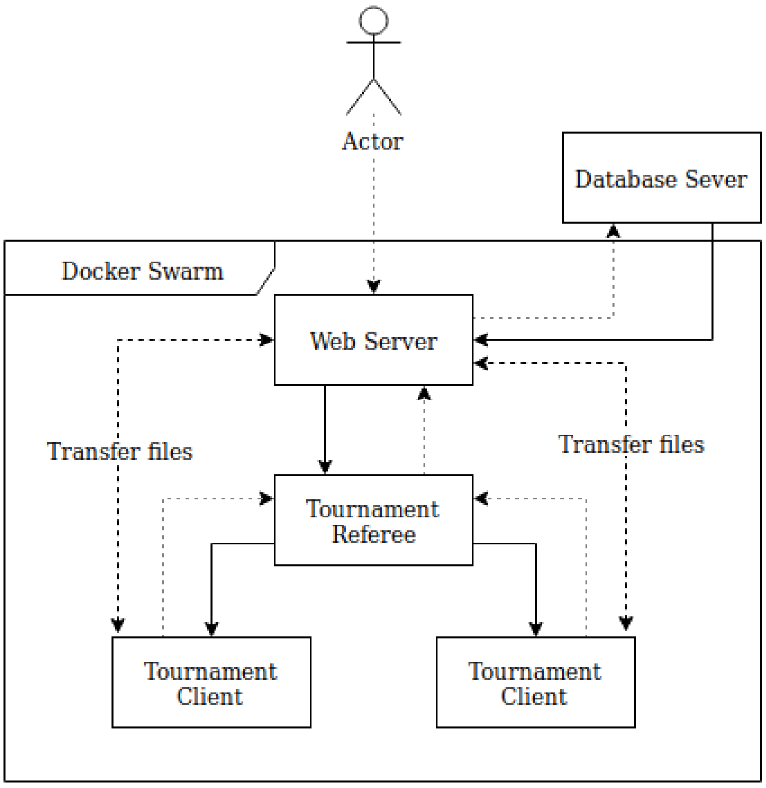

Figure 3.1: The layout and communication between different Docker components of the Cloud Tournament Engine

3.1 Docker and the Cloud

Docker is a large component of this project. The different features and services that Docker provides were utilized to develop the tournament engine into the CTE. This section will break down the different features and explain how they were used to create the CTE.

3.1.1 Docker Containerization

Docker containers are preconfigured virtual machines which can run on any system with Docker installed. Each major component of the CTE is packaged into a Docker container. Overall, the entire system consists of six different Docker containers, namely, the web server, database server, tournament referee, tournament lobby, tournament client, and the Othello tournament client. Each component of the CTE, running in its own container, allows each component to run on different machines that may be available throughout the cloud. More about these components will be mentioned in their respective sections below.

3.1.2 Docker Network

The Docker network is a feature that allows Docker containers to have their own private network to communicate. This functionality makes it possible to specify the properties for an overlay network which can only been seen by containers which run on said network [6]. This is used in the Cloud Tournament Engine for the different components in the system to communicate securely without having to configure ports or IP addresses which the components run on. For the CTE to function as intended, the only ports that need to be made available are TCP ports 2376, 2377, 7946 and UDP ports 7946, 4789. These are the ports that Docker network and swarm uses to communicate.

3.1.3 Docker Swarm

The Docker swarm is a feature that connects computers as nodes in swarm. It is possible to add many different cloud computers to the Docker swarm, these computers and their resources become available to use by any swarm manager. Docker swarm allows for a manager node to create a service, which is a task that needs to be completed [8]. For the purpose of the CTE, a service is a tournament client which needs to participate in a game. Services can be run with specific resource constraints, which makes it possible for users to specify their memory and CPU constraints for tournaments. Docker swarm handles load balancing so that each resource requirement is satisfied. If a service for a player is created and there are no available resources for the swarm to use, the player will wait until resources become available.

3.1.4 Communication between Docker containers

All components in Docker containers use the Docker network to communicate. The CTE needs to able to send log files from client and referee containers to the web server. In order to do this, a token system was created. For every match that is played, a random authorization token (currently set to 25 characters) is generated and passed to each player when the service is created. When the client container finishes its game, it uploads the game log via HTTP to the web server. If the authorization token is valid, the player log file is stored on the web server and the match is marked as complete.

3.2 Web Server

This is the largest component of the system and it does the bulk of running the CTE. The sub-components that make up the web server are the user interface, the tournament server, and the Docker listener.

3.2.1 User Interface

This section will cover how CTE user interface (UI) differs from the TE. The TE user interface allows the user to create, join, manage, or terminate tournaments. Users can upload their game schedulers, rankers, players and tournament configuration files. These files can then be used to start a tournament. It was necessary to change a few key aspects in the TE's user interface to reach the goals of the CTE.

Live Updating System

The CTE adds an overhauled live updating system. This allows the user to view which resources are available on the engine's swarm network in the form of connected nodes on the Docker swarm. Information about tournaments is broken down into a list of matches played, where each record shows the node where a player was executed on the swarm, the match outcome, and match log files.

Creating a Tournament

The nature of the CTE's error handling offers each tournament two additional constraints, the first constraint is how long a player can take to make a move. This can sometimes be specified in a configuration file for the IF, however, it is optional in some games and thus needs to be specified during tournament creation. Secondly, a user can specify how long a match can take to execute before the engine shuts down the match. This parameter is completely optional, however, it is useful in cases where a match could stall and impede a tournament.

Ranking System

The CTE allows for the user to provide their own ranking system for tournaments that are played. Included in the CTE is a default Elo rating system. When the user views the ratings of a tournament, they are able to view the Elo rating for that tournament, game wins, game losses, and a win-loss ratio.

Othello Tournament for RW314

The Othello game used in the RW314 course at Stellenbosch University is written in C. In order for this to be compatible with the CTE, a wrapper was written in the IF. This wrapper translate moves made by the C client to Java for the IF to validate and execute moves. This makes it possible to run the RW314 Othello Tournament on the CTE. Due to the fact that the structure of the RW314 IF client requires a C player to be uploaded, the user is given an option between RW314 and a general IF tournament. This choice is presented to the user in two places, the first is when creating a tournament, the user needs to specify that it is a RW314 tournament. The second is when the user uploads a player, the user needs to specify that it is a RW314 player and upload the compiled C file.

3.2.2 Tournament Server

The tournament server (TS) runs as the majority of the CTE's back-end. The original TE had a tournament server that was written in Java. The objective of the TS was to listen for database changes that happen when a user starts or alters the state of a tournament. The TE used a single loop that waited for input and would execute tournament functions that ran in separate threads. However, the following issues presented themselves:

- The separate threads were created by a listening loop that would fork from the main thread, but more often than not, the forked threads were not terminating correctly. This lead to threads rejoining the main process and executing new operations alongside other forked threads. This caused unpredictable behaviour.
- Sometimes the thread in the loop would fail on database calls and get stuck, and thus impeding the entire TE.

Initially, a lot of work was put towards debugging and finding faults in the TS in order to produce a well functioning CTE. This turned out to be very time consuming and the decision was made to completely overhaul the Java tournament server with a version written in Go. The following sections will cover why Go was used and the design decisions of the Go TS.

- Go has concurrency built into the core language. It is easy to create a concurrent process while using go-routines[1].
- Go does not need a virtual machine to run, it compiles directly to machine code, which decreases our overall container size because the JVM can be removed.
- Just like Java, Go is a well optimized language which is easy to read, it is also easy to extend for any additions to the system in the future.

The Cloud Tournament Engine's Go TS uses a different process in terms of execution in the sense that each new tournament that is hosted, starts a difference instance of the TS. This makes the server slightly more stable in cases where one server instance might encounter a fatal error; the others are left unaffected. The tournament server has a few sub-components.

[1]This is Go's improved version of threads

Ranker

The tournament server has a ranker component which is customizable and allows the user to implement their own ranking system. The user has access to three separate pieces of information to make use of in order to calculate the rank for a player. This is a list of all players in the tournament, a list of matches that were played, and the match results. The user can then use this to calculate a rank and display it on the ranking page for the tournament. The game ranker can be written in Go or Java.

The single program instance running per tournament enables the CTE to compile the ranker directly into the tournament server when using the Go ranker. This improves performance and simplifies the program because sockets are not required to communicate between two separate running programs. The Java TS required socket connections to communicate between the TS ranker and the ranker the user uploaded.

If the Java ranker is used with the Go TS, some performance is lost. This is because the TS needs to execute a Go ranker that acts as a wrapper which communicates to the Java ranker (which needs to be executed too) through sockets. An Elo ranker system has already been written for the TS, both in Go and Java.

Scheduler

Just like the ranker mentioned above, the scheduler is also uploadable in either Go or Java. This too, is compiled directly into the TS. The scheduler has more information made available by the TS for the user to use in order to create their own scheduler:

- A function that detects new players joining the tournament once it is running, which allows the user to add the player to the tournament in a appropriate place.
- There is also a function available which allows the user to stop a tournament at a specific point.
- The user has access to all matches played in the tournament, the players in the match, and the match results.
- A list of all players is available which allows the user to create a match between the given players, which will then be executed by the tournament.

This information is enough for the user to schedule matches, as well as decide how matches should be scheduled by the match outcomes.

Database Driver

The tournament server has a module with utility functions that controls how information is retrieved and added to the MySQL database. Most of these function calls were rewritten from the Java TS in the original Tournament Engine.

Docker Driver

This module contains the functions for executing and terminating Docker containers for matches. The Docker driver has a listener that waits for new matches, these matches are added by the scheduler that the user uploads. When a new match is detected the following steps occur to execute the game from start to end:

1. A docker network is started for the match.
2. A referee docker container is started.
3. The lobby docker container is started, this is given the game details and configuration file.
4. All the client container services are created, each given an authorization token to download the match player from the web server and later upload the player log file. The swarm then executes these services on available swarm nodes.

5. After all the players are started, the program goes into a state where it polls the database, checking if all the players have returned their token.
6. Once all the players have returned their tokens, the program removes the player services and shuts down the referee container.
7. The ranker can use the match results to apply the ranking system implemented.

It is important to note that all these steps happen for each match that is played. The function that detects when a match is meant to start executes these steps in a new go-routine, thus, matches can happen concurrently. It is up to the scheduler to add the matches to the queue, if all the matches are added at once, they will all be executed concurrently.

Input from the User Interface

The tournament server listens for changes to a tournament's state in the database. This allows the user to pause, resume, or terminate a tournament by toggling the corresponding buttons in the tournament management controls.

3.2.3 Docker Listener

The docker listener is a separate program which is written in Go. This program checks for changes in the Docker swarm and updates the database. The Docker listener has two specific functions, a listener that looks for services running on the Docker swarm, and a listener that watches for changes in resources available to the swarm.

The listener that monitors changes in services detects which player services are running and on which node the service is running on. This information is added to the database, and this allows the user-interface to display where a player is executed.

The second listener monitors the list of available nodes in the Docker swarm, when a node is added to the swarm, the number of threads and total memory can be detected. This information, along with whether a node is active or not, is added to the database. This information is then displayed to the user.

3.3 Tournament Referee

This section will cover the functionality of the tournament referee (TR) component and how it integrates into the system to form the CTE. The TR has has two components, the Ingenious Framework referee and lobby. Each component of the TR is built into it's own docker container.

3.3.1 Ingenious Framework Referee

The IF referee runs in a Docker container which allows for it to be easily started and stopped. This is convenient because each match that is played in a tournament has its own IF referee. When a new match starts, the TS starts a new referee container on the Docker network that was created for the match.

The referee container uses very little in terms of computation. Its role in a match is to communicate with all the players in the match and relay the moves a player makes to all the other players. Due to this, the referee container is not added as a Docker service to run on the Docker swarm. Instead, the referee container always runs on the same machine as the web server. A Docker referee container needs to retrieve the IF referee .jar file in order to start the game. Having the referee run on the same computer as the web server makes it convenient because the .jar file can be copied into the container without having to authorize a token and download the file through HTTP just as the tournament clients do. In the same way that files are copied to the referee container, the game log is copied out of the container once a match is concluded.

Running the referee container as a service on the Docker swarm was considered as an alternative to running it on the same computer as the web server. However, the latter was chosen as the method of implementation for the following reasons:

Tokens: If the referee is handled as a service, a token would need to be made to allow the uploading and downloading of the IF .jar file.

HTTP client: A Go program would need to be added to the referee container in order to send the authorization token to upload log files, this creates unnecessary programs running in the referee container. As well as contributing to the increased size of the container.

Idle Times: The nature of the IF referee means that a lot of the time, the referee container could be idle while waiting for a player to compute a move. This results in relatively low computation costs that does not drastically affect the computer running the web server.

All these factors contribute against running the tournament referee container as a service. Thus, leaving it running on the web server is the better option.

3.3.2 Lobby Client

The Ingenious Framework has a component called a lobby, the lobby creates a match with a given referee name, as well as provides a configuration file for the match. Similar to the IF's referee, the lobby is also built into its own container. The lobby container has two similarities to the referee container:

- Hosted on the same computer as the web server.
- A configuration file is required by the lobby which is easily copied into the lobby container after it is launched.

Unlike the referee, the lobby does not stay active for the duration of the tournament. The lobby container starts and receives the configuration file which is copied into the container. It then connects to the referee and informs it of the match details. Once this is complete, the lobby container stops. This short lived container does not use very much in terms of computation, and it is only started for a brief period. Thus, it can be considered ideal to rather have it run with the web server than as a Docker service running on the Docker swarm.

3.4 Tournament Client

The tournament client is a Docker container which controls the IF's match players. As mentioned in Section 3.2.2, the tournament client is started as a Docker service, which is executed on the Docker swarm to start computing moves for a match. Players communicate through the Docker network to relay moves to the tournament referee. The CTE has two different tournament client containers, one for general IF matches and another for Othello matches for the RW314 course at Stellenbosch University. Each client has subtle differences that will be explained in the following sections. Similarly, the containers share a program that finds and reports issues when a game is not proceeding as anticipated.

3.4.1 Error Checker

When running a tournament, an error may be encountered. This might be due to a badly made referee or a problem in the players which were made to run with the IF. Before the implementation of the error checker, a tournament match would suspend when an error was encountered and the tournament would not advance further.

In order for the error checker to function as intended, the user needs to define two constraints. The first constraint is the maximum time for a move to be made. The error checker is written in Bash, in order to enforce this constraint, the program checks to see if the player log file has changed every X seconds, where X is the maximum time to make a move. If the checker finds that the log file has not changed, it terminates the match by stopping the execution of the player. After the match is terminated, the incomplete player log file is uploaded via HTTP using the authorization token associated with the tournament client. When one player is terminated, the other players stop too. Each player's error checker uploads the incomplete log files using the authorization token granted when the service for the container was created.

The second specified constraint is the maximum time that a match can take to complete. If the checker detects that the match has gone on for too long, it will terminate the game and upload the player log file. The other tournament clients will do the same.

When the error checker terminates a match and uploads the player log file, it sets a flag within the database which indicates that the match ended due to not adhering to the constraints specified by the user. This flag is used to indicated to the user that the tournament ended abruptly. Incomplete matches are displayed in red on the dashboard page.

It is difficult to differentiate which user is at fault when a match is abnormally terminated. To remediate this, the match is restarted automatically. If a match fails three consecutive times, all the users in the match receive a strike and the match is disregarded. Once a user receives three strikes (nine failed matches), the user is disqualified from the tournament.

3.4.2 General Tournament Client

The general tournament client encompasses any game that can be played in the IF, excluding the Othello tournament client. This client is built into a Docker container, which is pushed and committed to the Docker Public Repository under the name MONKLEYS/CTE. The first part of the name indicates the username of the Docker Hub account, and the second part is the name of the container. This container is hosted on the public repository because it removes the need to add the container to each node that is added to the Docker swarm. When a Docker service is created for a player container it is allocated to an available node running on the Docker swarm, this node then connects to the public repository and downloads the latest version of the client. This means that any machine that needs to be made available for the CTE, only needs to be added to the Docker swarm and have an internet connection. It is possible to configure a private Docker repository if internet connectivity is a problem, however, it would be easier to update the container manually by importing the image from a local copy. Once a container is loaded onto a node, Docker only needs to connect to the public repository to check for updates.

When a match is started on the CTE, a Docker network is created and the TS creates a Docker service for each player in the match to run on this network. This service is given a unique authorization token, as well as a link to download the player .jar file. All information passed to a tournament client from the TS is done with the use of environment variables that are specified when the service is created. Once a service is allocated to a computer on the swarm, the container is pulled from the public repository, if it does not already exist. The client then uses the provided authorization token to download the player .jar file. Once this is complete, the player .jar file is executed and the player computes and communicates moves to the tournament referee through socket connections which connect through the Docker network. The log for the player is piped to a file inside the tournament client container.

Once a game is complete, the player terminates and executes a program written in Go. This program uses the authorization token provided to connect to the web server and upload the player log file via HTTP. Once the log file is uploaded, the token is marked as used in the database. The TS detects that the token has been used and sends a signal to terminate the Docker service running the tournament client. Once the Docker service is terminated, the container will terminate on the swarm node. This process happens for every player in a match.

3.4.3 RW314 Othello Tournament Client

This version of the tournament client operates in a similar way to the general tournament client, with one key difference. Just like the general tournament client, the Docker container has a name on the public repository[2]. The Othello game engine is written in C for the RW314 concurrency course project. To support the Othello game engine that is written in C, a wrapper was produced for the IF to translate moves made by a C player. The wrapper makes use of sockets in C and Java to communicate moves between the two programs. The Othello referee in the IF handles the game just like it would any other game that can be played on the framework.

The requirement of a C player means that the Othello tournament client not only needs to download the .jar player from the web server, but the C player too. Both of these files are downloaded using the authorization token provided to the container when it was created as a service. The rest of the container's functions happen in the same manner as the general tournament client.

[2]MONKLEYS/CTEO

3.5 Database Server

Just like the web server, referee, lobby, and client, the database server is hosted in its own Docker container. The database server is on a Docker network with the web server in order for the two components to communicate. The web server is the only other component that communicates directly with the database. Only a few changes from the TE were made to the database, this includes the addition of two new tables: TOKENS and NODES.

3.5.1 tokens table

In Table 3.1, it can be seen that tokens has an identifier column and six other columns:

- **id**: Used to identify rows in the table in the relational database.
- **token**: The authorization token issued to each player.
- **log**: The location of the log file for the match.
- **match**: Identifier reference to the matches table
- **player**: Identifier reference to the players table
- **complete**: The current match states, the states are as follows:
  - **-1**: Match failed.
  - **0**: Match in progress.
  - **1**: Match completed.
- **nodeID**: Identifier reference to the node table, indicating which swarm node the match was hosted on.

Table 3.1: tokens SQL table that was added, with example data

id	token	log	match	player	complete	nodeID
41	vFsqwHBMzKsQvpgWrJJjwXxCx	/oth1.log	31	13	1	4

3.5.2 nodes table

The nodes table has four columns and one unique identifier. These can be seen in Table 3.2, the following is what each column references:

- **id**: Used to identify rows in the table in the relational database.
- **host**: The name of the Docker swarm node that the service was hosted on. This name is the default name set by Docker after adding the node to the swarm.
- **cpu**: The number of CPU threads of the node.
- **memory**: The amount of total available memory on the node.
- **down**: This indicates the current state of the node:
  - **0**: The node is currently reachable, or turned on.
  - **1**: The node is unreachable, either due to loss in connectivity or the machine being turned off.

Table 3.2: nodes SQL table that was added, with example data

id	host	cpu	memory	down
1	reece	12	15.679	0

3.5.3 Other database changes

It was necessary to make a few minor changes to three preexisting tables, such as the addition of extra tournament ranking metrics, mentioned in Section 3.2.1, added to the TOURNAMENT_PLAYERS table. The TOURNAMENTS and PLAYERS table both received an additional column called TYPE, which indicates if the tournament and player table entries are for a RW314 Othello tournament or a general IF tournament.

Chapter 4

Testing

This chapter will cover the process followed for the results obtained from testing the Cloud Tournament Engine. The CTE consists of many parts, namely, user interface (Apache), database (MySQL), tournament server (Go), docker listener (Go), and the tournament client error checker (Bash). These are some of the most vital components of the CTE and they have all been tested to ensure that the CTE is performing as intended. Not only do these components need to be tested, but the overall integration between all the components needs to be tested too, including the ability to deploy tournament clients to swarm nodes.

4.1 Component Testing

This section will cover the testing of individual components. Some of these components can be tested using unit tests, other components need to be tested in a different way. How each component was tested, as well as the effectiveness of the testing method will be discussed.

4.1.1 User Interface

UI testing for this project involved running through each available feature on the UI and performing manual tests. After each action was performed it was necessary to confirm that the correct database entries were produced (more on database testing in the next section). The CTE takes in two types of inputs from the user, these include information written straight to the database and files uploaded to the engine. Both of these actions are vulnerable to different attacks.

For the database entries, any information entered by the user is only entered into the database through the use of prepared statements. Prepared statements force user input to be handled as the content of a parameter, this helps protect the system from users trying to inject SQL queries.

Users uploading files, such as C players, Java players, and IF referees, opens the CTE to major vulnerabilities because users could potentially submit a malicious program and execute it. To avoid this, a well configured .htaccess file prevents the user from exploring file paths on the system and potentially finding the path to an uploaded file. When files are executed, the execution happens within a Docker container, if a user were able to execute malicious code, the container would be isolated and the rest of the system will be left unaffected.

4.1.2 Database

The database was tested using the database driver mentioned in Section 3.2.2. Go has unit testing built into the core language. Unit tests have been created which loads information into the database and then retrieves the information. Testing using the database driver not only helps to determine that the database is running as expected, but it also ensures that the methods in the database driver of the TS are working as intended. The unit tests for the database driver achieved 100% code coverage.

4.1.3 Tournament Server

Part of the TS consists of the database driver, which was tested in the previous section. When writing the TS in Go, careful consideration was taken to compartmentalize the code in such a way that testing is convenient. The other parts of the TS include the ranker and scheduler.

Ranker

The ranker only has four functions that make up the entire class. These functions take input and have an expected output. Testing revealed an error in a function (getWinner()) which evaluates the scores in the match logs to determine which player wins a match. This has since been corrected and 100% code coverage has been achieved.

Scheduler

The scheduler has a number of utility functions, as well as functions that make calls to start and terminate docker containers. Testing the utility functions results in 60% code coverage. However, the functions that make calls to Docker in order to start tournaments cannot be used for unit tests, because, these functions are compounded by calling other functions which perform specific processes. These functions are evaluated during integration testing.

4.1.4 Docker Listener

In Section 3.2.3, it is mentioned that the listener has two functions. The first function monitors running Docker services and determines which node the service is running on. This is a very difficult program to create an automated test for because a service can run on any available node in the swarm, meaning the output is non-deterministic. Testing for this was done manually by starting a service, then comparing the output of the service monitor with the output of DOCKER SERVICE INSPECT. In all tested cases, this was determined to be working.

The second function of the listener is to monitor which nodes are connected to the swarm. This is also difficult to test because docker swarm nodes can only be added by connecting to another computer and connecting it to the swarm. For this, manual testing was performed by running the listener and inspecting the nodes it displayed. This was compared to the nodes that were actually connected to the swarm. The docker listener also retrieves the maximum memory and CPU threads that a node has, this was all confirmed manually.

While testing the functionality of the Docker listener, it was discovered that a computer in a swarm remains in the swarm even if it is turned off. This lead to the dashboard displaying a node as active even if it was switched off. This was corrected by adding a function that periodically checks the availability flag of the swarm node and updates the database accordingly. This resulted in the down flag being added to the nodes table in Section 3.5.2.

4.1.5 Error Checker

The error checker runs within the tournament client container. This program is written in Bash and has no formal methods that can be called. In order to test this, a new match is started and the referee for the match is forcefully terminated. The players are then left in an empty state where they receive no feedback from the referee. The error checker detects that there has been no changes in the game log files and it terminates the game. After the match restarts, the referee is manually terminated once again, this continues until a player gets disqualified.

The error checker was tested by starting a large number of tournaments concurrently. If this happens and not enough compute resources are available, some player services will start and other players in the match cannot start their service because not enough resources are available. This results in a system wide deadlock, however, once the error checker detects that a player has not made a move within the given time constraint, it terminates the player. This leads to the match restarting, which frees resources. Other tournament client services can then start, resulting in the CTE breaking out of the deadlock.

4.2 Integration Testing

The CTE is made up of different components which are isolated from each other. This makes unit testing difficult and thus integrationn testing needs to be used to evaluate how the different components perform together.

4.2.1 Concurrent Tournaments

It was necessary to test whether tournaments could execute concurrently. This was done by starting four different Tic-tac-toe tournaments simultaneously, three tournament had two players and the fourth had three. At the time of testing, the Docker swarm had four available nodes to use, with a total of 32 GB of RAM, and 24 CPU threads.

All the tournaments executed as expected. This was confirmed by logging into each node via SSH and inspecting running containers using Docker stats. With this, it was also possible to confirm that the Docker listener, mentioned in Section 3.2.3, was reporting the correct node where each player was being executed.

The three tournaments with only two players successfully completed a single match involving both players because the round-robin scheduler was used. The tournament with three players executed three matches successfully. The player log and match log files were examined and compared to the output reported by the CTE. It was confirmed that the results on the CTE reflected the results obtained in the game log files, this validates that the logs were being read correctly. This is just one of many experiments carried out to test the integration of components in the CTE. Throughout the development of the engine, integration between components was tested extensively.

The output of game log files is dependant on the moves made by players, which means the output is non-deterministic, thus, manual testing is the only way to confirm the tournaments are running as expected.

4.2.2 RW314 Othello Tournaments

The Othello tournaments were tested in the same way as concurrent tournaments. There are two types of Othello Tournament players, the random C player and the Distributed Tree Search players which were created by the students in the RW314 course. During the integration of the Othello Tournaments into the CTE, the random C player was used for testing. In order to confirm that Othello Tournaments could actually be used for the RW314 course at Stellenbosch University, the players created by students were used to test the engine.

A round-robin Othello tournament was created with three Distributed Tree Search players. When the tournament was created, a 4 second move constraint was set. This constraint was also set in the IF's configuration file. For resource constraints, the tournament was set to utilize 100% of a node's CPU and 8GB of memory. The same nodes mentioned in Section 4.2.1 were used. After the tournament was executed, the first match started, but only one player was executed and the error checker ended up terminating the match after the move timeout. The match was then restarted by the error checker, and it failed again. This continued until the match was disregarded.

This seemed concerning, however, after evaluating the four individual nodes on the swarm, their resources were as follows:

Node 1 Threads, 12. Memory, 15.679 GB.

Node 2 Threads, 4, Memory, 7.675 GB.

Node 3 Threads, 4, Memory, 7.669 GB.

Node 4 Threads, 4, Memory, 7.545 GB.

The resource constraints for the tournament were set as 100% CPU utilization and 8GB of memory, because of this, the services for each player in the first match was created and the first player was started on node 1. However, the other nodes could not satisfy the resource requirements for second players service, node 2 is shy of just 325MB of memory required to executed a player service. This behaviour is expected. Tournaments will fail to execute if resource constraints are defined but the required resources are not available.

To remediate this, the tournament's resource constraint was modified to use 7GB of memory. After making this change, all the players were able to execute on nodes 1-4. Once the matches were complete, the results from the player log files were compared to CTE's output, which was determined to be correct.

The RW314 Othello Tournaments are running as expected, for both random C players and Distributed Tree Search players which were developed by the students in the RW314 concurrency course.

Chapter 5

Future Work and Conclusion

This chapter will cover the future work mentioned in the report for the TE and future work that could be implemented in the CTE. A conclusion will be provided at the end of this chapter. Future work for the CTE entails: adding a more robust and user friendly front end, implementing continuous delivery, and using external storage options to make sharing of files between components more fluid.

5.1 Future work from the Tournament Server

The report from the original TE mentioned four implementations for future work [13]:

1. Spawning players and referees on different physical machines
2. Dynamic number of matches on a physical server depending on compute resources available for the server.
3. Allowing for player containers to connect to a different server where the player is hosted using sockets.
4. Visualization of live matches on the TE.

The first three implementations have been implemented in the CTE by making use of well compartmentalized components and the Docker swarm. However, the last requirement, visualization, cannot be implemented. Live visualization of games would require an observer to be added to the IF, as well as live feedback between the tournament server and players, or referees. The CTE only communicates with players and referees when transferring game and log files. Adding this feature would induce a large amount of network traffic on the Docker network created for a match. Therefore, it is not feasible to add live match visualization.

5.2 User Interface

The front-end of the Cloud Tournament Engine uses the same stack as the TE. Many changes were made to the TE to make it more user friendly, although, the UI could be more visually pleasing. Bootstrap was used in the TE and the CTE, however, React could be used to make the web interface even more user friendly and visually satisfying [11].

5.3 Continuous Delivery

As explained in Appendix A, the CTE requires the system to first have Docker-CE installed and Docker swarm needs to be initialized. Continuous delivery (CD) would allow the CTE to automatically deploy to computers in the cloud. This is difficult to accomplish because the CTE requires some software setup before tournaments can be created.

It is possible to make use of Terraform to implement CD [12]. Terraform provides the functionality for infrastructure to be written as code. This means the user can define a set of rules, which Terraform will use to create and launch computers (with specific hardware requirements) directly from a cloud service provider. Terraform allows integration with many different cloud providers, including Amazon Web Services (AWS), Azure, and DigitalOcean. In addtion to specifying the infrastructure to deploy, Terraform allows the user to indicate exactly what software to install on each cloud computer.

This functionality means the user could configure a massive swarm, potentially with hundreds of workers and mangers, without having to manually configure hardware. Not only does this allow for CD, but it also makes the CTE easily scalable.

5.4 External Storage Options

The CTE has different components that need to share game and log files. Currently, the CTE makes use of HTTP to directly download and upload files. This implementation works well, however, it requires a direct connection between components that would not otherwise communicate with each other. This could be remedied by making use of external storage options. AWS provides S3 buckets [1] and Azure uses blob storage [2].

The web server of the CTE could upload game files to these external storage options, then provide the information to download the files to tournament clients through the use of environment variables when a Docker service is created. The tournament clients can download these files directly from external storage. The same applies when a tournament client needs to upload a player log file to the web server. By making use of external storage options, the load from sharing game files is passed onto an external system which alleviates load on the Docker network. This will increase the overall performance of the CTE in the cloud, as well as make it more scalable.

5.5 Conclusion

The CTE had three requirements that needed to be satisfied: users need to be able to specify resource constraints for a tournament, the system needs to allow the addition of resources for players to utilize, and multiple tournaments need to be run concurrently. All of the above-mentioned requirements have been satisfied. Docker services make it possible to apply resource constraints to a container. Machines can be added to a Docker swarm as nodes, these nodes make their resources available for the CTE to use. A tournament server written in Go was implemented which runs a single tournament. Many tournament servers can run individually, which allows multiple concurrent tournaments to be executed.

This project has resulted in a well distributed tournament engine that can be scaled to allow many concurrent tournaments. Each tournament could use vast amounts of compute resources, entirely dependant on the machines made available to the swarm. The CTE can be further improved by implementing the features mentioned in Section 5.1.

Developing the CTE has been an insightful experience. The initial challange was understanding the original Tournament Engine. Different developers embrace different programming practices, conflicts in implementation and design can make understanding any foreign project difficult. For this reason, it is clear that software documentation is important. After extensive software archaeology[1], the TE's functions and what needed to be implemented to meet the requirements of the CTE, came to light. The CTE relies heavily on Docker, implementing it has been a great learning experience, not only about Docker, but also distributed computing, and the Go programming language.

[1]The study of poorly documented or undocumented software

Appendix A

Deploying the CTE

The only software requirement the CTE has is for Docker-CE to be installed. Docker-CE needs to be version 18.06.1 or higher. This installs all of Docker's components, including swarm and the ability to BUILD and RUN containers. Both of these features require some execution. The CTE needs to be started and the swarm needs to be initialized.

Running the CTE

The CTE is made up of components which are built into a number of Docker containers. In order to build these containers, the CTE's code needs to be pulled from a repository. The CTE can be executed by using the Bash script named START_CTE. This script will build and execute all the containers required for the system. The web server will be started on port 80 on the machine that the script is run. The engine will then be hosted and partially operational, the CTE will be able to accept all game files, including new players, referees, rankers, and schedulers. Tournaments can be created, however, they cannot start because the Docker swarm needs to be started for players to be executed.

Running the Docker swarm

The Docker swarm is required because it handles the player distribution to nodes, nodes are the machines in the cloud which are connected to the swarm. The swarm has two configurations, a *manager* and a *worker*. Managers are the nodes in the swarm that can distribute services for players to start be executed, this makes it possible for multiple CTE systems (which are handling multiple tournaments) to be running on the same swarm. Worker nodes are computers in the cloud that make their resources available to satisfy the constraints of tournament clients, in order to execute them. Manager nodes are also worker nodes, this allows the CTE to be executed on a single machine running the web server and the swarm.

The Docker swarm needs to be initialized, once the swarm starts, the machine that initialized the swarm has the ability to create *join-tokens* for adding other managers and workers to the swarm. Docker allows manager nodes to promote workers to managers. This ability to have multiple managers running web servers, each with their own database, a single shared database, or potentially a distributed database, makes the CTE very scalable with Docker.

As mentioned in Section 3.1.2, the only ports that need to be made available are TCP ports 2376, 2377, 7946 and UDP ports 7946, 4789. These are the ports that the Docker network and swarm use to communicate.

Bibliography

[1] AWS S3 buckets. https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingBucket.html/.

[2] Azure blob storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[3] Docker. https://www.docker.com/.

[4] Docker-CE. https://docs.docker.com/install/linux/docker-ce/ubuntu/.

[5] Docker network. https://docs.docker.com/network/.

[6] Docker overlay network. https://docs.docker.com/network/network-overlay/.

[7] Docker resource constraints. https://docs.docker.com/config/containers/resource_constraints.

[8] Docker service. https://docs.docker.com/engine/swarm/services/.

[9] Docker swarm. https://docs.docker.com/engine/swarm/.

[10] Ingenious Framework. https://bitbucket.org/skroon/ingenious-framework.

[11] ReactJS. https://reactjs.org/.

[12] Terraform. https://www.terraform.io/.

[13] Linde, H. Tournament Engine for the Ingenious Framework. pp. 19–20.