Game Contest Server

Developing the Back-end

Alex Sjoberg
Department of Computer Science & Engineering, Taylor University, Upland, Indiana 46989

Introduction

The Computer Science and Engineering Game Contest Server is an ongoing department project started in the fall of 2013. The goal of the project is to create a web site that allows professors to host contests for various games such as checkers, chess, and battleships. Students will then be able to develop and submit automated players for these contests so that the professor can play them against each other and determine who can program the best player.

Previous Work

During fall semester 2013 Dr. Jonathan Geisler and students in his Multi-tier Web Application Development class worked on building the basic foundations for the site using the Ruby on Rails web framework.

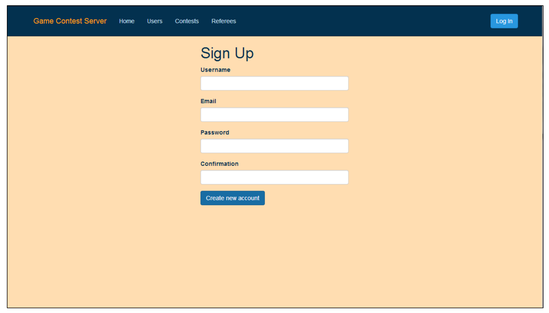

Figure 1: The site's sign up page, created by students in COS243

Foundational things such as the ability to log in to the site and create, view, edit, and delete various database records such as players and referee were already completed, allowing us to focus more on the site's unique functionality.

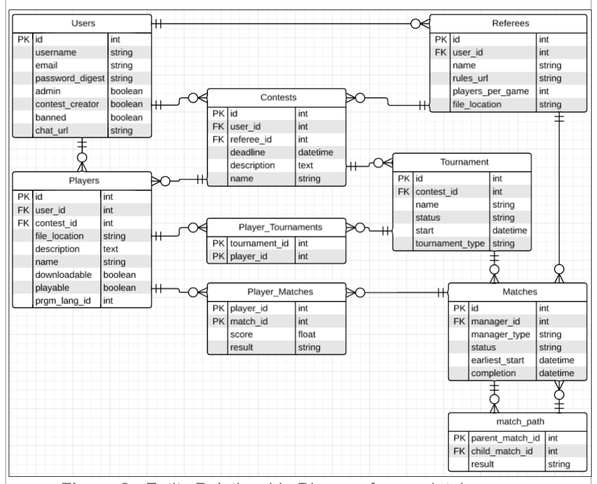

Figure 2: Entity Relationship Diagram for our database. The tournament and match_path tables were added during January to better reflect the back-end.

Back-end Goals

This January my primary focus was on getting our server to be able to actually run matches and gather results to write back to the database. Both Dr. Art White and Dr. Stefan Brandle had pre-existing "referee" code to control the game logic and players written by previous classes that they wished to use with our server, so one of our goals was to make integration with our system as easy as possible. In the case of Dr. White's introductory programming class we even wanted integration with our server to be practically invisible to the students.

We also needed to be able to create a system that was game and language independent, as we want the server to be able to be used with any new games that the department may add in the future.

The back-end was primarily developed using Ruby, as this provided the easiest integration with the pre-existing front-end functionality.

How Do We Communicate?

We began by making a simple player and referee to both test and confirm our approach to communication and to serve as an example for the bare minimum required for players and referees to work with our server. We decided to have the players and referee communicate via sockets both with our server and with each other.

Sockets are supported in most programming languages and have the added benefit of allowing us to run the games over a network if it ever becomes desirable that different players reside on different machines. Having players and referees communicate with each other through sockets also has the advantage of making the players language-independent from the referees as well as from our server.

Running a Full Match

We next created a "match wrapper" class which would be capable of running a match with any arbitrary players and referee. This class encapsulates all the interaction we need to do with the player and referee files directly and thus defines what those files need to do to work with our server. It works as follows.

We begin by opening a socket on an available port which we will use to receive information from the referee. We then use system calls to fork and start the referee as a separate process, using a command line argument to tell it the port we expect it to talk on.

The referee must then send us the information for the port that the referee will expect its players to connect to it on. The wrapper then starts the players in the same manner, telling them the port on which they should connect to the referee.

From there, all communication takes place between the referee and players until the referee sends us back the results.

We leave all communication regarding how the game is played up to the referee and players so as to maximize our compatibility with different games. This also means that players only need to interact with the server on initialization, which makes writing server connection libraries easier and minimizes the impact on student code.

We later realized that when the referee was reporting back the results we had no way of knowing which player corresponded to which database record. Thus when starting players, we now also tell them a name which the server will recognize them by, which they must then communicate to the referee when connecting. This ensure that when the referee says that "Player Foo" won, we report it as a win for the right player.

Front to Back Operation

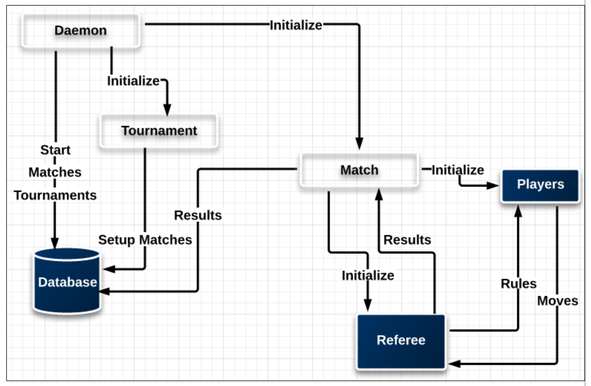

Figure 3: Illustration of program flow, starting after a user has set up a tournament using our web interface

After a tournament has been created using the front-end, it will be noticed by a daemon we have set up to continually query the database for tournaments and matches which are ready to be run. When a tournament is found, the daemon fires off a script to create all of the individual matches that make up a tournament, which varies based on the tournament type (single elimination, round robin, etc). These individual matches will then be picked up by another script, which gets the necessary player and referee information from the database, initializes a match wrapper as described earlier, and writes the match results back to the database.

Originally a tournament would simply run all the matches itself rather than create the database records and let another script actually run them. This made it much easier to keep track of cross-match information like the bracket for single-elimination, but eliminated the possibility of delaying the start of certain matches (such as watching the final round at a special time) or allowing individual challenge-matches separate from a tournament.

Adding the second script for running individual matches solved these problems, but necessitated the addition of the "match_path" table to our database as shown in figure 2 in order to allow for single elimination brackets.

Checkers Library

With a full working system from front to back we turned our attention to adapting Dr. White's checkers program to work with our server. As stated, we wanted to be as un-intrusive as possible and wanted students to be able to develop and test using the offline referee with minimal changes needed before uploading .

The referee only interacted with the player through a single function call, thus all we had to do was add our initial set up code via the importing of a library we wrote and then change that function call to instead send the request over a port.

Similarly in the player code, we have the player import a library to handle connecting to the referee and use a listener to wait for the referee's request to call that specific function before handing control over to the student's code.

Future work

- Though parts of the back-end were designed to work with more than two players, much of it was not. Some minor modifications will need to be made for running games with three or more players.
- Players and referees should be tested when they are uploaded to the server to ensure they work with our back-end correctly.
- Currently all uploaded files must be executable. Though this is okay when working with Python or Ruby, it means that compiled languages must be compiled on a machine similar to our server environment. Ideally we would compile the programs on the server.
- Since we are simply running executable code that students upload, security risk is high. Various security measures should be added to ensure students cannot upload harmful code.
- Challenging other players to an individual match should be possible given our architecture, but will require significant front-end development and a several back-end changes.
- It should be possible using X11 forwarding to allow running matches to be displayed on a remote screen. This would allow checkers tournaments to be broadcast as they have been in the past.
- The ability to log and replay matches should be possible with some more information-gathering during the match

Acknowledgements

- Dr. Jonathan Geisler for the original idea and requirements for the project and for his help in understanding the mysterious inner workings of Rails.
- Dr. Stefan Brandle for his endless knowledge of Unix and help understanding socket programming.
- Dr. Art White for providing us with his checkers referee and players so we could modify them to work with our server
- Nathan Lickey and Nate White for help setting up our VM and creating these posters.