Abstract

My senior project is a lightweight game server that facilitates two-player games. The server maintains a database of user’s usernames and passwords, available games, the state of each game, and a list of active matches between players. Communication between the client and the server is facilitated on the server side and relies heavily on multithreading to make the server’s connection to multiple clients as fluid as possible. The game server instantly sets up games between random players or specific ones via a search option. In order to implement a particular game, one must provide that game’s set of rules to the server. To keep things simple, I will be using the game of checkers to test the server. However, ideally, any turn-based game could be played on my lightweight game server.

My project covered variety of topics that I have studied in my time at Yale including: object-oriented programming to create instances of games, relying on well-designed databases to store player and game information, and use a variety of cryptographic primitives in order to maximize the security of the database and deter cheating in game play. In addition, my project employed several facets of coding that I had not had the opportunity to study, most notably networking, multithreading, and socket programming. Through new and old I think I was able showcase the many things I have learned from the Computer Science department at Yale. This database will hopefully provide a basis for other games in the future and allow many users to play along!
**Client- Server Model:**

The code for the client and server was written using the Python programming language. I first worked to study the client and server model of typical game servers—finding that a dedicated server, a server running on a dedicated machine, is the best, albeit most costly, approach to implementing a server. However, because I will only be playing lightweight games like checkers I decided I could use my computer as a dedicated server. After some research, I decided to use the socket library in python in order to establish the network connection. In addition, the socket library is very bare bones, which allowed for a large amount of flexibility when designing the server and client.

![Client-Server Model Diagram](image)

**Figure 1:** The Client-Server Model

**Client Implementation:**

In order to minimize the complexity of the client-server model and decrease the risk of attacks from the clients, the client file (client.py) was implemented with only the bare
necessities needed to communicate with the server—as such, the client contains less than 20 lines of actual code. The client first connects to the server on port 5004 using the built in functions in the socket library and the server’s IP address. Because the sockets are blocking, the client is unable to send data and receive data at the same time. In order to circumvent this, I used multi-threading to split the client into a listening thread and a sending thread. The listening thread constantly waits for the server to send information and then prints it to the client’s screen as it comes up. The sending thread waits until the client enters data before sending it to the server. This allows the client to have a constantly updated screen while simultaneously being able to respond to prompts from the server.

Figure 2: The client’s welcome screen
Please Enter Your Username: test
Please Enter Your Password: test
Welcome to Checkers, test!

Instructions: Please choose one of the following:
- Log In
- Create Account
- New Game

Figure 3: Post Log-in greeting screen

Instructions: Please choose one of the following:
- Search User
- Find Game
- Search Game List

find game
waiting for game...
waiting for game...
waiting for game...
waiting for game...
0 1 2 3 4 5 6 7
B 0 B 0 B 0 B 0
0 B 0 B 0 B 0 B 1
B 0 B 0 B 0 B 0
0 1 0 1 0 1 0 1 3
1 0 1 0 1 0 1 0 4
0 W 0 W 0 W 0 W 5
W 0 W 0 W 0 W 0 6
0 W 0 W 0 W 0 W 7

WHITE - Please Enter Move:
5 5 4 6

Figure 4: Waiting for a new game to start and the beginning of a new game.
**Server Implementation:**

Like the client, the server uses python’s built in socket library in order to function. After connecting to the database (described below), the server sets up a socket and listens for incoming connections at port 5004. Currently, the server is implemented to listen for up to 100 connections, but this can be increased or decreased at the discretion of whomever is running the server. For each client that connects, the server creates a new thread, using python’s threading library, which allows the server to fluidly handle multiple clients. Once a client thread has been created, the server starts running the thread. The server facilitates communications between the client and itself by sending prompts to the client and appropriately handling their response. To simplify testing, the server currently only allows checkers to be played, but more game modes can be added simply by importing the rules into the server file and specifying which game mode you want activated. When a game is started, either through searching for a player’s username or finding a random game, one of the threads host the game while the other thread is put to sleep. If one of the players disconnects or the game finishes, then the sleeping clients thread and the hosting thread are started up again and resume activities from the main menu.
**Figure 3:** The server first starting up, which shows it connecting to the database and listening for new connections. We can see that two clients, both also from my laptop, connect to it.

**Databases:**

I used postgresql to manage the project's database. I designed each table in the database to be in 3NF format. In order to access the database via python code, I used psycopg2, which greatly simplified access to the database as a whole and access to various entries in the tables. When the server is first started, it connects to the database stored on my computer and prints an affirmative message if the connection to the database was successful. Each client thread stores the database cursor, which gives it access to the various tables. The following are the tables of the database:
Security:

Besides the normal game functions that check if moves are valid, there are several security measures in place to prevent malicious use of the server. The server limits how much data the client can send and receive, which helps prevent memory attacks. This also prevents attacks on the database, as input that is too long is not accepted. New entries into the database are checked to ensure that duplicate entries are not added, and each new entry for both games and users is assigned a unique ID that can be used to individualize players and their games.

Checkers:

The checkers game is implemented in a separate python file, Checkers.py. Each checkers game is an object that contains the board in a dictionary. During game play, each move is attempted on duplicate board (copy of the dictionary), which tests if the move is valid. If it is a valid move, which includes multi-jumping pieces and kinged pieces, then the move is performed on the real board. If the move is invalid, then an error message is sent to the player and another move sequence is requested.
Adding a New Game Mode (Adding Stefan Doehler's Game to My Server):

Tasked with adding another computer science student's game, a version of Yahtzee, into my server, I had to make several small coding changes in order to allow clients the ability to play this new game. I first included the game rules, written by Stefan Doehler, which are located in a python file called k.py into my server file. I then changed several aspects of the rules to allow them to use sockets. Then I simply created a list that contained threads of clients who were looking for a kismet match. By simply creating a new prompt that asked if the client wanted to play kismet or checkers, clients can be easily sorted and game play can be started up very quickly. In order to show the difference in code between just a checkers server and the checkers/kismet server, I submitted two versions of my source code—server/client/checkers.py for just the checkers server, and server2/client2/checkers/k.py for the duel game server.

```
Instructions: Please choose one of the following:
-Log In
-Create Account
-New Game

log in
Please Enter Your Username:
test
Please Enter Your Password:
test
Welcome to Checkers, test!
Instructions: Please choose one of the following:
-Log In
-Create Account
-New Game

new game
Instructions: Please choose one of the following:
-Search User
-Find Game
-Search Game List
-Back

find game
Please enter game type - Kismet or Checkers:
kismet
waiting for kismet game...
waiting for kismet game...
waiting for kismet game...
Player 1 Score:[None, None, None, None, None, None, None, None, None, None, None, None, None, None, None]
Player 2 Score:[None, None, None, None, None, None, None, None, None, None, None, None, None, None, None]
```
**Figure 4:** A client logging into the server, finding and waiting for a new kismet game, and the initial game screen!

**Conclusions:**

The lightweight game server consists of only ~1000 lines of code, but its implementation really pushed the boundaries of my coding experience. I had no previous experience in networking, multithreading, and very little experience coding in python, but I learned a lot as I went through the semester. The final product takes a small amount of memory and is flexible enough to host hundreds of game types and clients at the same time. Adding in new games was easy and fun—it simply required a couple minutes of work on the server in order to enable a new game mode for players! This project was a lot of fun and really boosted my confidence in my coding. I would like to continue working on it and improving it as a side project for the future.