ECE 421 Assignment 5 Part 1

Winter2019_Group4:

Nathan Klapstein (1449872)

Tony Qian (1396109)

Thomas Lorincz (1461567)

Zach Drever (1446384)

Table of Contents

Design Questions **1**

References **4**

Design Questions

How do I handle starting and serving two different games?

There are several options: multithreaded servers, scalable microservices, or a combination of the two. Given the requirements of this assignment, a simple multithreaded HTTP server should be sufficient to handle the requirement of “10 users on independant machines”. Using this model, there should be more than enough computational power to support 5 concurrent games using one physical server running 10 threads.

How do I start new servers?

For our design we have decided to build a RESTful API using Puma - a concurrent web server for Ruby (the default web server for Ruby on Rails). Puma will spawn new threads to respond to requests, therefore we are not required to manually scale our application by spawning new servers as microservices using a platform such as Docker. We briefly considered using an RPC client and server, however, due to our design choices for Assignment 4, and mainly our storage of game state as a stateful document, we decided it would make more sense to build an HTTP server to transfer state between servers and clients.

How can a client connect to a game?

One client will need to start a game by using a “host game” button in the main menu. This game will be assigned a unique ID, and the game state information will be stored in the server with an empty slot for the second player. Other clients will need to search for games which are started, but do not have two players playing. Once a player selects to join an existing game, they will fetch the game data from the server and then they can also begin playing pieces.

What happens when only one client connects, what happens when three or more try to connect?

When only one client connects, they will be allowed to place the first piece in the game, then they must wait for another player to place the next piece in the game. This game will then be added to the list of games waiting for another player to join.

There will be a race condition for multiple players selecting a game to join. If two players select to join a game, the player that selects the game first will be allowed to join the game. The second player will be served an error code alerting them that this game already has two players.

What synchronization challenges exist in your system?

Our design is based around the idea that the state of a game is always accessible to all players, and each turn is a discrete update to the server. The servers communicate with a MongoDB instance that will handle CRUD operations atomically.

Because the game state drives the synchronization once in game (blocking a user from playing when it is not their turn), there are limited inherent synchronization issues with gameplay. The only synchronization issue that may exist is if two clients select a game at the exact same time before the server can update the state of the game to indicate that a client has already connected. We will need to be able to detect this scenario, and implement a tie break method.

How do I handle the exchange of turns?

Once a client has made a move, they will then need to pend an HTTP request to get their opponents move from the server. Because we want a responsive UI, this will likely be the most challenging issue to solve. There are two ways we could solve this: a non-blocking poll, or seperate the UI and HTTP client into two separate threads. Likely the best solution to this problem is to separate the HTTP client and the UI into two separate threads. This will allow the user to use all elements of the GUI except for the game board while they wait for their opponent to play.

What information does the system need to present to a client, and when can a client ask for it?

We will need the following GET endpoints for our server:

- GET/game_state&<game_id>
    - Returns game state for the game_id
- GET/new_games
    - Returns games that only have one user connected
- GET/in_progress&<user>
    - Returns games where <user> is a participant
- GET/leaderboard
    - Return the leaderboard of wins/losses

The client should be able to make a request to any of these endpoints at any time.

What are appropriate storage mechanisms for the new functionality? (Think CMPUT 291!)

In assignment 4 we use a Hash structure for games which contains the state of the game (including game board representation as a Matrix). The most logical way to store this information is therefore to JSONify these structures and store them in some sort of document storage. We have selected MongoDB as our NoSQL document storage. mLab is a MongoDB

company that supports database-as-a-service and has a free service tier that will provide more than enough storage space.

What synchronization challenges exist in the storage component?

For any storage the largest issue is write operations to the same data occurring at the same time. Because the game application is state driven this shouldn't be an issue during game play. Two clients should be prevented from posting a move to the server if it is not their turn. The leaderboard could potentially have this issue if game results are being posted concurrently, or reads and writes are being performed at the same time. Fortunately MongoDB has concurrency locks for this exact issue:

"*MongoDB uses reader-writer locks that allow concurrent readers shared access to a resource, such as a database or collection, but in MMAPv1, give exclusive access to a single write operation.*"

What happens if a client crashes?

The rest of the system will not be affected. The game state will be saved based on the other client's current game state. The other client can exit the app and attempt to rejoin the game so long as the other client is still in the game.

What happens if a server crashes?

On the next HTTP request that each client makes to the server, the TCP layer will throw an exception. We should be able to catch these errors, and the online services should fail gracefully.

During an online game, a server crash will exit both clients to the main menu. The last played game state will still be stored in our document store. Once the server is running, the two clients can attempt to restart the game by choosing "load game" from the main menu. The players will use their usernames to connect so that each client is treated as player 1 or player 2 as they were before.

What error checking, exception handling is required especially between machines?

All network requests should be wrapped in a "begin/rescue" block . If the HTTP request fails, we should be able to catch the exception and provide an error message (where appropriate) to the client. Retries for network requests should be considered.

Do I require a command-line interface (YES!) for debugging purposes????? How do I test across machines? And debug a distributed program?

Yes, having a CLI can be helpful. The CLI can help to eliminate layers of abstraction between the underlying model and presentation layer, allowing the programmer who is

debugging to be closer to the model. This eliminates the presentation layer as a possible source of error. We will continue to use the CLI that we built for Assignment 4 as a debugging tool

The best way to debug a distributed program is to connect the server and clients through localhost. By doing so, you can have all components on the same computer and more closely monitor the interaction between the clients and server.

What components of the Ruby exception hierarchy are applicable to this problem?

Ruby has a NET:HTTPException class which is subclassed to provide more descriptive exception types. Most of these exceptions will likely be important for error handling network requests. In addition, there are network related error numbers (errno) that will be important to test for. These include: Errno::EHOSTUNREACH, Errno::ECONNREFUSED, Errno::ENETUNREACH, Errno::ETIMEDOUT.

Describe the three most important personas utilized by the group in the design of Assignment 4 and explain how your design evolved to accommodate these essential requirements.

As a user, I want to play connect 4.

As a user, I (kind of) want to play Toot and Otto.

As a developer, I want the pain to stop.

References

Official MongoDB FAQ's on concurrency:
https://docs.mongodb.com/manual/faq/concurrency/