Assignment 5 - Distributed Connect4, Design Questions

How do I handle starting and serving two different games?
Two different games can be started and served by providing a stateless (RESTful) API for all game actions. All information required to serve a request from a player is contained either in the request itself or in the database. This architecture makes it easy for the client to quit then pickup games, because the server does not maintain a persistent connection between players.

How do I start new servers?
There is an endpoint dedicated to creating a new server (i.e game). This endpoint is called in the user challenge screen provided in the UI. Basically, it initializes a starting game state and corresponding empty game board in the database. To start the database server, a system admin will have to start it using the server.rb file.

How can a client connect to a game?
Since we are using a RESTful API, the client simply gets the game information from the server then he/she can start playing. Moves are recorded by sending the updated gameboard back to the server at a “take turn” endpoint.

The client is connectionless, our version of a “connection” is simply querying the server via RESTful API.

What happens when only one client connects, what happens when three or more try to connect?
Again, since our architecture is connectionless, it can handle arbitrary amounts of users. It also scales very well because we kept the game mechanics client-side (yes, at the MASSIVE cost of security but this was a conscious decision :))

In fact, our database is truly the only bottleneck to scalability, since we can load-balance the servers.

What synchronization challenges exist in your system?
One synchronization challenge was how we can wait for the remote player to take their turn. Since we have a connectionless system, the server is unable to notify the client when it is their turn.

We could have used a websocket, but a more RESTful way of implementing this functionality is to have a simple polling loop on the client side, calling an endpoint telling them when it is their turn.

How do I handle the exchange of turns?

Adding to the information presented above, while the client waits for an opponent's move (i.e the client is viewing an active game where it is not their turn) they execute a simple polling loop that indicates if it is their turn for a particular game_id. If it is their turn, then the client fetches the game an allows the user to input the next move. During this time, the remote player may also be polling the server.

Once the user completes their move, they send an update of the game board to the server, which will unblock the polling loop on the opponent end.

And so on...

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

The system needs to convey game information, such as ongoing games, users, previous game outcomes, and error messages. The client may ask for gameplay information from within the various game menus whereas the client may only dismiss error dialogs.

In addition, the client should be able to request general about/help information from all windows in the game.

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

A database is an appropriate storage mechanism. The storage mechanism needs to be able handle multiple synchronous requests, enable persistent storage, and be able to configured to restart itself in the event that it crashes. For this project, we decided to use a Mysql server to cover this functionality.

What synchronization challenges exist in the storage component?

Certain operations must use transactions to be atomic. Since we are not partitioning our database, we do not have any consistency issues. If this system were to scale, we would have to start thinking about the locality of each game's data.

What happens if a client crashes?

Nothing, everything is stored server side and it is a connectionless system so this is no different than a client simply leaving the game. As such, if a client crashes, the state of the game will still be saved in the server. Then, when the user logs back into their account, the state of the game will be the same as when it crashed. The client does not pushes all game state information to the server at once. That way, if the client crashes in the middle of performing a move, nothing will be pushed to the server, and the game state will be unchanged.

What happens if a server crashes?

If a server crashes then the player will be unable to play the game until the server is rebooted. Once the server is rebooted the player will be able to resume the game because the database was flushed to disk. The state of the game will remain unchanged while the server is crashed, and since players only communicate with the server and not the other player directly, then there will be no chance of a client having a game that is updated ahead of the client.

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

We have implemented a seamless exception handling structure wherein server-side exceptions get caught, serialized and passed back to the client over XMLRPC. Therefore, the client must know what exceptions may be thrown by the server, and handle them appropriately client-side.

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

Yes, a command-line interface is useful for debugging. Testing across machines cannot be easily automated. Instead, tests have been written to cover all gameplay logic, such as win-analysis, gameboard manipulation, etc, on a single machine. Once the gameplay logic is sound, the communication layer can developed separately and tested manually. Logs are helpful for debugging distributed programs -- all exceptions caught, and even basic logic can be written to logs to help debug during development and integration.

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

The two largest sources of exceptions for our program include handling UI and client-server communication. Both of these features have their own exception types and therefore standard Ruby exceptions are not as common. Our program also implements custom exception types that inherit from the StandardException ruby class. Therefore, the main components of the Ruby exception hierarchy that are used in our application include: StandardError, ArgumentError, and SystemExit.

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.

Children between the ages of 6-12, busy college students and a system admin.
Seeing as there would likely be a wide age group for our target audience, we designed our game with multiple computer difficulty settings. This was to provide each type of player with an enjoyable experience. We also realized that some players, especially the bored college students

will not have time to play a full game of Connect4. As such we have designed our game such that a player can disconnect and resume a match at any time. We also designed a very simple and intuitive UI so that players of all ages would be able to use it. For more hardcore players, we also have a leaderboard where players can see their standings and play competitively. We also designed our application with the idea of a system admin in mind. The system admin would essentially be in charge of starting a maintaining the server.