ECE 421 Group 5- RIley, Aaron, Jayden, Shang

- • How do I handle starting and serving two different games?

  There should be a virtual server, that has all the methods that hosts a game instance, that is spawned by the “real” server. This virtual server, is launched whenever there are greater than 2 clients connected in the queue to the “real” server. Each virtual server will run on its own thread.

- • How do I start new servers?

  There is a physical server which runs 24/7. When 2 client are connected (in a queue), it will spawn a virtual game server that will run on its own thread, hosting the game between the two clients. It will then remove the clients from the queue.

- • How can a client connect to a game?

  Ip address and a port that is hardcoded into the client file. Client cannot choose which game he/she joins so it automatically connects to a server that is not full.

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

  If only 1 client connects, then game will not proceed until it gets a second connection to the server, in which after that the server will no longer accept connections and the game will proceed. Server signals both client a “start token” once connection is full. In the event when multiple clients connect to the server, clients are paired together based on a first come first serve basis, and the any single remaining client, will not be able to proceed with the game until another client connects.

- • What synchronization challenges exist in your system?

  We want the game state to be synchronized with both clients and only one client should be executing their turn at a time.

- • How do I handle the exchange of turns?

The server will handle which player's turn is currently in progress and we will only accept inputs from that user during their turn. We will ignore inputs from the other user server side and possibly client side as well.

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

  The system should be presenting the client with the most up-to-date state of the game. The client should only need to know what the board looks like and whether it is their turn for the game. Also, a client should know the status of the other client that is connected and notify them if their opponent leaves.

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

  Databases. Possible tables: Active Game table, Game Record Table, Token Locations

- • What synchronization challenges exist in the storage component?

  The main challenge would be modifications to database at the same time and ensuring that the modifications occur properly/atomically. Ie, if inital value is 0, and user1 adds 5 and user2 adds 6, we want result to be 11 instead of say 5 or 6.

- • What happens if a client crashes?

  The game should end if a client crashes. Alternatively we could have the server wait until the client reconnects and store the current game state for when it starts up again.

- • What happens if a server crashes?

  The world ends and so does your game. The server should be updating the database storage after every turn so upon restart it will reload the last saved state of the game.

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

  We want to make sure the connections are established correctly before initiating the game. As well we should also be checking and updating the state of the game so that both clients are synced up properly whilst playing.

- • Do I require a command-line interface (YES!) for debugging purposes????? How do I test across machines? And debug a distributed program? • What components of the Ruby exception hierarchy are applicable to this problem?

  Command-line interface would be convenient as we can print values on to the terminal for debugging purposes, as opposed to looking at the UI and trying to figure out what went wrong. Testing across machines, we can try by running multiple terminals on one machine (mocking clients), that are connected to the server (also run on local machine) through localhost connection. Exception hierarchy that are relevant to this assignment include SystemCallError::Errno which will allow us to deal with network connection errors (ECONNREFUSED). Additionally, ThreadError is useful for handling exceptions related to spawning multiple servers running in separate threads.

- • 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.

  Robust Exception Handling - In assignment 4 we made sure to do necessary Exception Handling. We focused on sanitizing user inputs such that no malicious/unexpected input is passed into the game before initiating an instance of the game.

  Modularity - In assignment 4 our program was written with modularity in mind. The loose coupling of modules makes it easy to use in different contexts, for example in a client - server architecture.

  Generality - In assignment 4, our program was written to be as general as possible. This means it is possible with very little modifications/no modification, to accompany a variety of different game modes such as TOOT/OTTO, Connect4, Gomoku etc...