Group 4 Members: Isaac Supeene, Braeden Soetaert

Design:

Master server runs on server computer with functions to create a new virtual server (game) and to access the leaderboards.

Clients spawn a new game by talking to the master server. Clients can also get a list of currently running servers and choose a server to join. Players beyond 2 should be rejected from a server.

All servers should have at least 1 client connected to them. If a server does not have a client connected to it, it should actually be dead.

Need to ensure successful client-server communication and handle errors here.

Clients will need to identify themselves to the server with a username, for leaderboards to be stored properly.

Leaderboards, game results, and saved games stored in SQL database.

To enable two-way communications, clients will also run an XMLRPC ‘server’ (the View for the client) to receive XMLRPC callbacks from the game server.

Game server will actually be listening on three ports and will have 3 XMLRPC “servers”. 1 port for each controller server (1 per player) and 1 port for the game server itself. Game server will be the point of contact when a new game is being set up or a client is joining a server. The controller server will be the main point of contact for normal game operations (playing the game). The server side will also be running a client to perform callbacks on the Views using the MVC pattern.

How do I handle starting and serving two different games?

Client can start a game by contacting the server computer and getting a server created with the port for the current game. Multiple games can be started by making new servers through the server spawner on the server computer.

How do I start new servers?

We could run a persistent program on the server computer that clients call to get a new game. This program would spawn a server for the game and return the port that the client needs to talk to for the game.

How can a client connect to a game?

The server spawner can maintain a list of live servers on the server computer. A client can then receive this list and choose a game to join. Games can be made with a name so that users can easily identify them. A client will also connect to a game when they create a new one via the master server. Clients will need to identify themselves to the game server in order to receive callbacks.

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

When only 1 client connects, that client will wait for a second client before starting the game. If it is not the start of the game, (i.e. a client disconnects) then the player who is still connected will win. May later introduce a grace period for disconnects or an option that lets remaining player wait for reconnect.

If more than 2 clients try to connect to a game, the first 2 will connect and the others will be rejected by the server and not added as a view or given a controller.

What synchronization challenges exist in your system?

Synchronization challenges that exist in our system are keeping the games up to date between clients including the list of games that can be joined. Another synchronization issue is that of the leaderboard/rankings staying synced between all clients.

How do I handle the exchange of turns?

With our controller class, turns are known based on game's context and the controller's number. The client that has the correctly numbered controller that is equal to the game's current turn has the current turn. 1 controller will be given to each client in a 2 player networked game. A client is notified of its turn via the turn_update function in the views that the game model calls.

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

The server needs to present the game board, current turn, player controller and other game data to the client such as if a move was invalid. This will mainly be provided to the client via callbacks using MVC where the server, which contains the game model, will call turn_update on the client's views with the relevant information. Clients should also be able to ask for this information from the game every so often in case the communication method failed. The client will be able to ask for some information through their controller server, and other information via the game_server for the current game.The client should be able to ask for this information via the controller whenever it needs as long as a game is currently running on the server on the port the client is trying to contact. Client must recognize if no controller server or game server is actually present .

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

An appropriate storage mechanism for the new functionality would be to use a SQL database to store the leaderboards. The SQL database could also be used to store saved games for future loads. This way the saved games have a similar storage space. Different parts of the game context could be stored in different columns in a saved game table.

What synchronization challenges exist in the storage component?

We have to synchronize updates to the SQL database so that if 2 games try to update the database at once, that the database receives both updates and that the updates do not overwrite or corrupt one another. Also need to synchronize loads of games as a game will be deleted when loaded. If 2 games attempt a load at the same time, only 1 game should get it and also only that 1 game should be deleted from storage.

What happens if a client crashes?

If a client crashes or disconnects then the current game will end with the other user winning. In the future we could add functionality to allow a player to rejoin a game if their client crashes. If the game was only initialized and was waiting for another client to join the game,

then the game will end. The server will detect if a client crashes by occasionally calling the client during periods of inactivity.

What happens if a server crashes?

If a server crashes then the game will not be counted towards the leaderboards. Also if a server crashes then the client will notice as it will periodically check for server presence during periods of inactivity if a game is currently supposed to be running.

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

Main problem that needs to be handled with error checking and exception handling is communication between machines. Inter-machine communication is prone to both failures and delays and these need to be handled appropriately. If there are failures or lengthy delays then the machine will need to resend the info if the other machine requests it or if the sender detects an error.. Also the receiving machine will need to be able to handle receiving multiple sets of the same data in the case of a lengthy delay where the data was sent again but then the original data showed up later.

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

Yes we require a command-line interface because it allows us to have automated tests and also allows us to better test our backend without issues with the GUI factoring into play. We can test across machines by monitoring the outputs on both machines. A test program could also do cross machine communication to tell each when and what to input to the game.

Debugging a distributed program is significantly more difficult than debugging a non-distributed one. It can be debugged by remembering that there are two sides, and unless it is a communication error, then the error likely occurred on the side it was reported on. So first off, check for communication errors whenever a bug shows up. After you are sure the communication is good, check for the bug on the affected side. Finally check on the other side if the bug can not be found on the affected side. Debugging a distributed program will be much faster if you know all the components you are debugging. For instance, if the gui suddenly starts displaying a bad looking board, you know which function on the client side, and which function on the server side are the important ones for you to look at.

The best way to debug a distributed program through development is to develop the ends first, and then tie the ends together using network communication. In this way, both ends can be tested in isolation first for functionality, before communications comes into the mix.

- **What components of the Ruby exception hierarchy are applicable to this problem? Illustrate your answer. Consider the content of the library at:http://c2.com/cgi/wiki?ExceptionPatterns Which are applicable to this problem? Illustrate your answer.**

From the ruby exception hierarchy, the only real ones that comes to mind are StandardError and NoMethodError. StandardError is applicable because we may need to develop our own exception class to deal with errors using XMLRPC. The NoMethodError also may come into play with XMLRPC if we are trying to call a method on a server from a client and that method does not exist on the server.

With regards to the exception patterns, CatchWhatYouCanHandle is important as the communications code should be catching only failed attempts and if it fails enough times, it should be throwing a different exception that it won't catch so that the client can notify the user of a failure to contact the server. LetExceptionsPropagate also factors into this fairly well for the same reason as CatchWhatYouCanHandle. DesignByContract is always useful as this is how we design our programs. The ExceptionReporter pattern may also be useful in terms of easily reporting exceptions to the client, either via a GUI window or the command line.