Assignment 5

How do I handle starting and serving two different games?

All existing games are stored in the database along with which two players belong to that game. When a client decides to start a new game, he chooses the other player he'd like to play the game with. The client then sends this information to the server which will create a new game instance and store this information in the database. Each client, when in the main games window, will periodically poll the server for all games that user belongs to, and the client will display these games in a list. The user can then select that new game from the list and the board (initialized in the initial call to the server) will be displayed on the client. When a client user makes a move, it will send a request to the server, which will then update the database for that game. The server will send game model info to be displayed on the client whenever that info is requested (periodically by the client). This allows for support for an infinite number of games and games that can be played when only one user is online at a time (or saved for later when neither player is online).

How do I start new servers?

Simply launching our server application on a specific machine, and providing a port number which will handle accepting all incoming packets.

How can a client connect to a game?

Client will first need to provide the server IP and portno. Also will need to login using a username and password. This username will be queried in the games database to display all games currently being played by the user. User can select which game he would like to play, then make a move.

Similar to words with friends, we will allow players to play games with users not currently connected to the server, and the user will be notified the next time they log in.

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

Each single client will see a list of all games which the currently logged in user is playing. If he wishes to start a new game he will see a list of users (online or offline) who he can play with.

If he starts a new game or opens a game then he will send a message to the server to add a new database entry, or to query the database for the game board info of the current game, the server will send a copy of the board state to be displayed, and if it's the user's turn to make a move he will be able to make a move.

This will be the view for all potential players. One two or three connections won't make a difference. The server is not providing an always-on connection between the 2 players playing the current game, but instead providing a means for the client to send moves to the database, which can then be queried by the other user.

What synchronization challenges exist in your system?

There will be two major synchronization challenges with our system. The first will be interaction from the client and server. We will try to minimize this by storing our game state only on the server side, and having nothing being stored on the client. That way if any messages are lost between client-server, the worst that will happen is a player may have to remake a move when he sees that his board has not updated.

There may also be a synchronization problem if a user makes multiple moves rapidly without the server updating current user's turn. Extra precautions will need to be added to ensure only the first message is accepted as a valid move

Another synchronization challenge will be from the server storing information in our database. Since multiple games may be happening at the same time, extra care will need to be taken on how we structure our database, and how our server ends up accessing our database.

How do I handle the exchange of turns?

All of the logic will be handled on the server. All clients will simply contact the server periodically to find out the current game state (and in that will be the current player's turn).

Eg. PlayerA will make a move, and ClientA will then send a message containing the move to the server. ClientA will message server and get the current game state, which has been updated to let PlayerA know it is no longer his turn. ClientB may then contact the server getting the current game state, and will now dispaly that it is PlayerB's turn. ClientB will not necessarily be connected at the same time, and may potentially connect to the server at a much later time.

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

The system needs to present the current game state to the client. The client can ask for the game state whenever it wants, and we will set this to happen periodically.

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

Database.
Table1: Games

GameID | Players <Vector> | GameState (Board, current player's turn) | Winner | GameOver?
Table2: Users Players | Passwords

What synchronization challenges exist in the storage component?

Since we will have the server update the database, and each game stored as a different entry, there will be no instances of multiple updates to the same entry at the same time. This shouldn't be a problem, because if the game is working as intended, no one can make a move to the same game at the same time.

What happens if a client crashes?

Nothing really, everything is stored server side. The worst case scenario is that the players move is lost, assuming the crash happened before the info was sent to the server, but after the player made a move. In which case when he launches the program he will need to remake his move.

What happens if a server crashes?

Nothing again, with the exception of the fact that the game will no longer be playable without restarting the server. Since we're not using an always-on connection when the server crashes the worst case scenario is a single turn was lost in the same way it could occur if the client crashed.

What error checking, exception handling is required especially between machines? Do I require a command-line interface (YES!) for debugging purposes????? How do I test across machines? And debug a distributed program?

Server-Client will need error checking to make sure messages have been successfully sent, as well as server-side exception handling to ensure the message is valid (eg: make move).

We will be testing most of the server-client functionality via localhost, and for multiple machine connections we will use minimal test cases.

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.

There exists an exception Errno::ECONNREFUSED which is thrown when a server connection is refused (the server specified by the ip and port cannot be connected to). We may implement a loop that retries the connection a given number of times, notifying the user if all retries fail.

ExceptionsCancelTransactions: On the server side, if an exception occurs while updating the game, we will need execution to continue normally and in a valid state. This may involve a database rollback, and we'll also throw an XMLRPC::FaultException with a given exception code and message which can be caught on the client side to indicate the current turn was not executed. The client side may also execute requests in a loop that retries the current request a given number of times as long as the Exception is thrown. This also follows the ConvertExceptions pattern since whichever exception occurs on the server side may potentially be converted to an XMLRPC::FaultException.

BouncerPattern is still being used (from our initial implementation) for when a user tries to make a move when it is not their turn.