2D-3D-Tic Tac Toe

Michael Lawrence

McCartor McKinney

Andrew Connolly

Our project is a game of 3 dimensional tic tac toe in projected into a two dimensional form with up to four players. The win condition is the same as normal tic tac toe with 3 in a row meaning a win. The game allows users to play with their friends or random players by creating a public or private lobby, or joining an open one via a lobby viewer. This is accomplished by utilizing a server / client relationship, where the server runs, operates, and makes changes to the games and the client gives the user the ability to interact with the server. This includes requesting information or submitting changes. This two way communication is done with a network application protocol we developed to ensure that when a message is sent or received, the proper action is taken.

Server and Network Protocol

To run the game you must start the server with Python3 server.py

And you must each player must run Python3 OtorioGame.py ServerIP

With ServerIP being the IP of the server

The main functionality of the server is a series of commands.

The major functions that the main server has are

newGame

joinGame

gameList

leaveGame

All of these commands are fairly self explanatory and the only things of note are that each expects a response of some form back right now that is limited to either the expected output in the case of leave game or a good or bad based on if the command worked or not. To expand on this a little more when the server is started it loads up a thread that listens for incoming connections when it receives one it spawns a new thread to deal with it; this thread remains in place until it has dealt with the client connection. Inside the thread it accepts the connection and deals with its request if it attempts to join a game or create one it creates a game in the case of create game or join the game in the case of join game. Both of these pass on the needed information to the gameThread which then communicates with the client so that it knows where the game is. There are currently no limits on the number of games that can be created I would need to test it before I can say for sure on how many would be a good number of games to run on the server.

Inside the game the major commands are.

getPlayers

submitText

getBoard

leaveGame

submitMove

startGame

joinGame

These commands don’t expect a response back and instead wait for the game server to send one back. Each of the previous commands is exactly what it sounds like the only notes to make is that joingames allows a player to reconnect to the game as well. getPlayers sends in a list of the players in the order they have joined to the person who requested it.leave game does not delete the person it simply marks them as inactive and they could rejoin the game later.

The protocols that we developed have two major forms: the first is any message sent to the main server before joining a game; this is just a standard message that is expected to be less than 1024 bytes long. This message will have a format of: name requestType extra extra extra with each part of the message being separated by a single space. The reason for this is a large part of the server needs the name so it was arbitrarily put at the front in hindsight we probably should have put requestType there though. Once the main server gets the message it will send a reply to the connection this reply consists of either the word good or bad this is to let the client know if a request worked or not nothing more as of this moment. To join a game the client must provide the port number of a listener thread to allow the game to communicate with them. The reasoning behind this is to ensure that each connection does not need to be kept open indefinitely. To communicate with the game there expected format of the message is different in this case due to the fact that it's harder to tell the size of the message since chat messages could be rather long as well as a few others the format is that two messages are sent the first one is the word sending # the number represents the number of bytes that the message will be in length while the message length is less than this it will keep listening for more of that message.

Client Portion

When the client is run, It starts by showing the user the first screen. Here, the user must enter a name then choose if they want to create or join a lobby. If the user chooses to create a lobby, They will be taken to a screen to enter the details for the lobby. This includes the name of the lobby, if it is public or private, and if it is private, the password to join the server. Once the server is created, the data input will be sent to the server, creating a new thread to manage this individual lobby. The game will then go to the wait screen where they will wait for the lobby to be full (four players). If the user goes back at this point, they will leave the lobby. If they leave the lobby and they are the last player in it, that game will be removed from the list. If the user p-icks join lobby instead, the user will see the lobby browser. From here, the user can see all of the lobbies, how many players are in each, and if it is public or private. If a private lobby is selected, the user will be asked to enter the password. Once a lobby is joined, the user will go to the wait screen. Once the lobby reaches four players, the game will begin. On this screen, the user will see the rules, gamboard, and text box. On the user's turn, they will pick up a piece and place it on an open place. The user then confirms the move, sending it to the server which will update all the other players and see if they won with that move. The next player will then take their turn and so on and so forth till there is a winner. The chat box allows players to talk to each other by submitting text to the server and the server sending it to all the clients. Once a player wins, the user is taken to the win screen where they can choose if they want to play again or quit.

Network usage

The main thing that could be improved is that we create a large number of sockets only to send a single message. The reasoning behind this is that it allowed a reliable way to communicate back and forth between the server and client. Due to a time issue the rejoin game was never implemented though so this ended up being pointless to a degree. Traffic wise the game didn’t need a massive amount of information. Due to the fact that we only needed to send small strings back and forth to communicate very few of the messages were more than 100 bytes long the only message that could easily pass this size was the chat message. The game was also never tested over a network due to the fact that none of us have a network with multiple systems. We are positive that it works over loop back at the very least though.

Challenges

(McCartor McKinney start)I’m gonna list the challenges I have faced with this project.Basically the challenge of this project was that we decided to split it up into individual parts. While it makes sense to split it up it also made it harder to work on since I had less of an idea of how the other parts were implemented and they had less of an idea of how my parts were implemented. The only other thing was that figuring out how to make it so that a person could get disconnected and still rejoin a game was a massive challenge and in the end my solution to it wasn't the best though I have a better idea of how to do it in the future (McCartor McKinney end).

(Andrew Connolly) The hardest part of this assignment for me was ensuring that all of the users and threads could properly communicate with each other. Because of how we organized it, It was challenging to find what method is needed for each situation. It also required a lot of communication between us to confirm that what we are sending is expected. Sadly, we did not properly write out a full reference sheet for the protocol, leading to a lot of confusion and discussion. I know my group mates and I would have made the structure more organized if we had the chance to do it over. I also had some trouble thinking of the logic on how to chain all of the screens together, leading to the games at the end since we already had this portion written. I required me to chain the calls to create the screens, returning “done” when finished and passing that string all the way back to the first screen. From here the main loop starts. It would not be a super big deal, but we allowed the user to go back and forth between screens. It was also somewhat tedious setting up all of the rectangles to show information to the screen. For example, when the game is started, we have an issue where the turn marker is incorrect until the first move is done.

(Michael Lawrence) This was the first project where I had to program my own GUI using a library. I am used to using visual studio’s window form gui creation tool. I created the center Otrio game board such that it is possible to resize the game, move the individual pieces, and easily use functions to manipulate the board state as needed. This was by far the most challenging part on my end to figure out. I also implemented the client side of the networking project. For the game management, I ended up using a listener thread that connects to the server and stores commands into a queue which is then managed by the game thread. This part to implement was also tricky, but it works and works pretty efficiently.