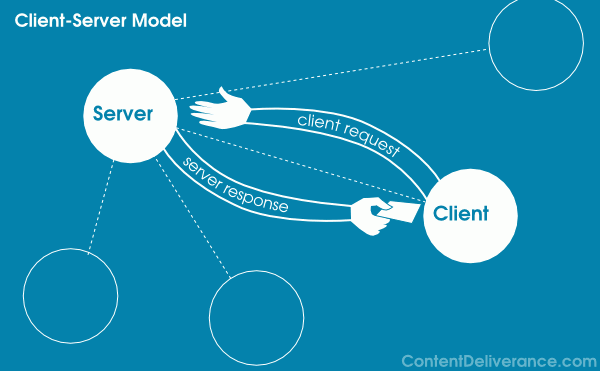
Our design team chose to use a client-server model architecture as the major software architecture in our game design. A client-server model architecture partitions tasks and workloads between the server and various clients. The server will receive messages or requests from a client, and all the computation will be handled by the server thread, which means the client thread will only have the functionality to send messages to the server and get a reply from the server to update the GUI for the individual client who sends the request. The client-server model is more often used in web programming which has a real physical server to run the backend program and has the web browser to execute the client program.



In our design, we will not have the server backend program run on an actual server, but we will design the game to use the client-server model using various threads. We will treat each player as a client, and the game model as a server. The server will only communicate with one client at a time, depending on which player’s turn it is. This means that all the clients will be blocked from sending a request to the server except the one client who’s turn it is. After one client sends a request about what they want to do in the game (e.g. move, fire), the server will update the game information and send the updated information back to the current client. After every turn, the server will give the client a copy of the updated board to use, in order to keep each client up to date on the current model of the game.

There are several reasons why we chose to use a client server model as our software architecture rather than using a MVC(Model–View–Controller) design which all of us are familiar with. The first reason is that we found this made sense logically based on the structure of the game in the abstract. Several different player instances communicating with a single board model is very similar to several clients communicating with a single server. By making the game model the server and each player a client, the remaining design of the game naturally structures itself around this central design, where no players communicate with each other and only communicate with the model itself. The second reason for choosing a client-server architecture is that it was a challenge and something that deviated strongly from the other groups, which we all found appealing as a simple way to challenge ourselves and do something a little bit more interesting than the standard.

Although we chose a client-server architecture as our main architecture, we’ve only implemented it after the actual game instance is running. For our main menu that is loaded before the game begins (the menu sub-system) we’ve used more of a view-controller architecture, where a single class runs and monitors each menu, and when a new menu is needed the controller class calls the class needed to display that menu. In this way, we can keep the code running the graphical user interface separate from the code running in the background, while keeping everything concise and easy to implement.