# Tic Tac Toe – Design Document

### SE 3313 – Final Project

### Shayaan Ali, Davinderpal Cheema, Alexander Powell, Riddhi Sharma

### <https://github.com/ShaynAli/Ticta>

# Introduction

Tic Tac Toe is a game traditionally played between two players on a 3x3 grid. Players take turns placing X and O respectively in the empty spaces on the grid. This project takes the basics of tic tac toe and builds an application to allow players to play against each other over the internet. It will take the concept of tic tac toe and stretch it to support any number of players on any board size (n x n). The application is built using a client-server architecture where clients connect to the server and play each other with the server acting as the messenger.

# Game Flow

The following is a sample of a possible game flow:

1. The server is started.
2. The server opens up a new thread for accepting client connections, and a thread for matching players to games.
3. Some number of clients open the client side application.
4. The clients connect to the server and are put into a list of clients by the server thread.
5. The matchmaking thread creates a game thread and passes in a list of clients that will participate in the game.
6. The game thread builds and sets up the game.
7. The server passes messages to each client letting them know that they have been matched into a game and the state of the game (number of players, board size, turn order etc.). This is done through a thread built specifically for managing message passing to the client.
8. The client thread listens for messages from the client and updates from the game. When a message is received it is passed on to the other party (ex. client -> client thread -> game or game -> client thread -> client)
9. When a player enters a turn into the GUI the turn is sent to the server.
10. The server computes the game logic for each turn and passes the board updates to all of the clients participating in the game.
11. Steps 9 and 10 are repeated until the game ends.
12. When the game is finished the server sends the result of the game to the clients and then terminates the game thread and adds the clients back into the list of available clients.
13. Clients will then be matched into a new game if the new game button is pressed. This repeats steps 4 through 13 until the server is terminated. Otherwise the client either sits idle or is terminated by the user on the client device.
14. When the admin types the stop server command into the server console, the server is terminate. All server threads will end and the program will exit even if there are games in progress.

# Design

## Runtime Environment

The runtime environment chosen was python. This was chosen because all group members have experience building server-client applications using python. The client was also built using python in order to make it easier to build; most of the client socket code is similar to the server code. Python threads are easy to use which helps with the implementation of the project and the threading module includes many synchronization objects such as semaphores and events.

## Server Design

In the design phase of this project the server was designed to compute all of the game logic. This is to prevent a client from maliciously sending false game logic to the server. The client is only responsible for sending the player moves to the server. The server verifies if the move is valid and if so it computes the game logic and sends board updates to the other clients participating in the game.

The server was originally designed to have two types of threads where one would control clients connecting to the server and add them to new games. The other thread would control a single game and take messages from the client, compute game logic and pass messages to other players in the game. This would make the server design very simple and would be easy to implement. This had the drawback of being difficult to read and understand. As the application grew the design changed to make the code more modular and reusable. The new design would be able to potentially support any game with the only change being the implementation of the game itself.

In order to facilitate the new design there were several threads added to the server. First there is a server console thread that takes inputs from the server admin to control the server state, this thread also has the unique ability to kill all of the threads on the server. Then a server thread that accepts new clients and add them to a list of connected clients. The server also has the capabilities to support several matchmaking threads that will each create new games and put add clients into the game. For each client the server runs a tread that waits for updates from the client and relays them to the appropriate thread. The game is controlled by a game controller thread that uses the client threads to send and receive updates from the players of the game.

Each of the above threads are necessary. The client tread needs to be constantly listening for messages from the client because the client could send at any time. The server thread needs to be constantly listening for clients because a new client could connect at any time. Because of this the matchmaking threads always needs to be running so that they are ready when players connect to the server. The game threads are created dynamically every time a new game runs; because there can be any number of games and because the game moves can be received by the client at any time during the game, the game thread needs to be running for the duration of the game. These threads spend most of their time blocked waiting for messages from the clients but all of these thread types are required to build the game with a more modular structure.

## Client Design

The client is designed and built to be a thin client, the client does not perform any game calculations. The client is simply a user interface that accepts inputs from the user and converts them into game moves. The game moves are then sent to the server; the server handles the game logic and sends board updates and game updates to the client. The client uses a few threads, it has a thread for passing messages to the server, another thread that listens for messages from the server as well as a thread for handling GUI events.

# Synchronization

The clients are synchronized by the server. The clients connect to the server and the server handles all message passing between them. If clients play out of turn then the server will stop them. If the clients try and make an invalid move the client server will stop them and tell them to make another move or wait for their turn. The clients cannot become out of sync because all turn and game logic is computed on the server and the server facilitates all messages that are passed between the clients.

On the server side there is only one resource that is used by multiple threads and that is the list of connected clients, in order to synchronize this list the server uses a semaphore to control access to the resource.

# Termination

The server is terminated is handled through the console. The server admin can simply type a stop command to stop the server from running. This terminates all threads that are running on the server and boots any clients from in progress games. Termination in python is not as simple as calling threads destructor in C++. Because threads cannot be killed externally, the server requires a termination event object that is shared by all threads. When the event is signaled all threads terminate themselves. This is a small drawback to using python as a runtime environment but is easily handled.

# Iteration

The original designs for the server was much less modular and the design would not be reusable without significant modification. The first iteration of the design had only two types of threads on the server side: a thread to accept client connections and another thread to control the game logic. The final version of the server has five different types of threads and each thread serves a purpose that does not depend on the implementation of the other threads on the server. By doing this bugs can be isolated more easily and fixed. This also allows for parts of the server to be easily modified to fit future purposes such as a checkers game server or a monopoly game server. When one part of the server is modified the other modules that depend on it do not need to be changed, they operate by knowing the methods that the other module will use and what each methods will return.

The object oriented style of the final version of the project took longer to make but it produced a better result. The code of the final version is much more readable than the first and all of the classes perform fewer functions working together to make the game work instead of a few classes doing all of the work. The first version had a few points of failure for that would take the entire application with it. In the final version some threads can fail without taking down the server.

During development a throwaway prototyping model was used to create the final version. There were several iterations in order to create code that is easy to read, efficient and modular. The first iteration was not very object oriented and code acted more like several scripts running on different threads. The next version was more object oriented and focused on how the specific parts of the server and client would need to be built in an object oriented style. The next iteration took the knowledge from the first iteration and tried to extend it to how the server and client classes to work together with each other. Then in the next iteration we focused on how the client and server would interact with the game and the user interface. Finally taking forward the knowledge of the last iteration the project was started again from the ground up in order to remove lingering mistakes from the prototype phase and finally create a product that met the goals of the final project.

# Discussion

What did we learn?

The labs were very helpful in learning about how multithreaded programming works. Having never worked with synchronized threads before the learning curve was steep. The labs help ease into using different synchronizing objects such as semaphores.

What did we think of the labs for this course?

The labs were good overall. The content was very programming heavy which was helpful in taking the operating system principles and applying them to real programs.

What did we think of the course overall?

The course was very well taught, better than the other courses this semester. The content was interesting and a little challenging but nothing crazy. It was a very productive learning experience overall.