Report by: Jason Ho Jia Sheng, 22360143

Q: How would you scale up your solution to handle many more clients?
Use more threading/multithreading, create a thread for each client that connects to the server. It does have its own limitations like overloading the CPU of the server if there are too many threads/clients using the server at the same time and high traffic. Therefore, I would use a better CPU like this one or AMD threadripper CPUs with better processing power and more cores(for handling multiple programs simultaneously) for my device that will be hosting the server. Less chance for overloading the CPU of the server and can handle many more clients.

Add more servers. I would also use the load balancer that is used a lot in parallel computing to scale up my solution to handle many more clients. Using this, we can distribute the requests to all available server configurations or specified servers capable of fulfilling them. Helps ensure no one server is overloaded with clients which can degrade performance. Then, we can handle many more clients.

I will also use the pub/sub framework. Allows loose coupling and scaling between exchange of messages between clients. Clients can operate independently from one another and remove service dependencies. Also allows dynamic networks to be constructed at a scale without overloading our current servers, resulting in more “memory” to handle more clients.

Q: How could you deal with identical messages arriving simultaneously on the same socket?
We are using TCP in our implementation and TCP can deal with identical messages arriving on the same socket simultaneously. The end point of a connection is socket. We can identify the socket by 2-tuple (IP and Port). To identify the connection(or link) we need the 4-tuple (IP of client, client port no., IP of server and server port no.). If the same messages arrive simultaneously, they will be distributed by different servers, then the server IP will be different and both messages will be differentiated. In the TCP header, we have its source port, so that 2 or more identical messages are distinguishable because they will be sent by **different machines or ports**. Therefore, TCP can handle identical messages arriving simultaneously on the same socket.

In socket programming, when two messages, whether it is identical or non-identical, arrive simultaneously, one of the messages is inserted into the waiting queue at the receiver's end and it keeps on waiting till the ACK packet of the current message is sent to the sender's end. When the sender receives the ACK packet, a table is updated at both ends, which contains the details about the messages, like message ID, size, sending time, receiving time that are already sent to the receiver by the sender. Then, the message in the waiting queue is received by the receiver, and the details of the message are loaded into the system. If the two messages are identical, I don't want the receiver to receive a duplicated message/data, I can handle it through programming and refuse or discard at the receiver's end. The receiver sends an ACK packet that contains the information about the duplicity of the message and the message is not received at the other end. The table at the sender and receiver's end will be updated. Can also ask the sender to send again. If identical messages arrived from the **same device/person** simultaneously on the same socket then we can choose 1 over the other and discard or ignore the other one.

Q: With reference to your project, what are some of the key differences between designing network programs, and other programs you have developed? When designing a network program we must note that it can expand very easily without disruption of service to all current clients connected to the server through assigning a thread for each new client connected. Other programs may not need this feature. Network Programs have a lot of extensibility. Traditional programs have limited extensibility.

When designing network programs we have a central node, the server which acts as a switch or hub for the clients connected to it. Means it will be easy to troubleshoot and debug. Traditional programs may not have this and it will be harder to troubleshoot.

Network programming is used to set up a network of connecting computers, routers, switches, wires or any other network-able components. Network programming includes developing software for clients, servers, and writing codes so that the client and server interact more effectively.

Network programs are codes to run a network. Important to know about commands for devices like netstat, ping, and ipconfig. Network programming includes many tasks like distributing resources over the network, finding the best routing protocol, knowing firewall laws, and bandwidth allocation for efficient use.

General or traditional programming only involves the user interacting with the device whereas network programs are written for end users (the clients), hardware, routing, and quality of service for the clients.

Q: What are the limitations of your current implementation (e.g. scale, performance, complexity)?
Time complexity could be shorter especially when I broadcast my messages to all clients connected to the server. My current complexity is like no. of times need to broadcast * O(total no. of clients to the server). Maybe use hashsets? I also have tier4 limitations as I did not implement some of them like when a client joins in the middle of a game, they won't have the full updated board as spectators. For tier3, when a client joins in the middle of a game, sometimes they have the wrong color of the tokens. Certain edge cases for disconnection not addressed. I would have liked to set a timer for clients to join instead of just starting immediately at 2 players.

Q: Are there any other implementations outside the scope of this project you would like to mention?
I created a countdown timer between games but I removed it out since it is not needed.

Q: Any other notable things to discuss. Tiers 1,2 and 3 are done, refer to README.txt. Tier1 is done through multi-threading, assigning each client a thread and then using a list/queue to keep track of turn order. The reset function in tier2 is done through checking if the queue is empty or only 1 client is left and then rebroadcasting everything again. If active player list is more than 4 then we assign the clients that joined recently to the spectator list. Tier3 is done like removing disconnected player from queue etc and allowing clients to join mid-way through the game by making them join the spectator list.

Overall, although I did not finish the project, I found it to be interesting and would have liked more time to complete it. More support and guidance would definitely be appreciated for the project especially for the early tiers. I also feel some of the tiers could be broken down into smaller easier segments especially tier1 as I myself was stuck on it for weeks.