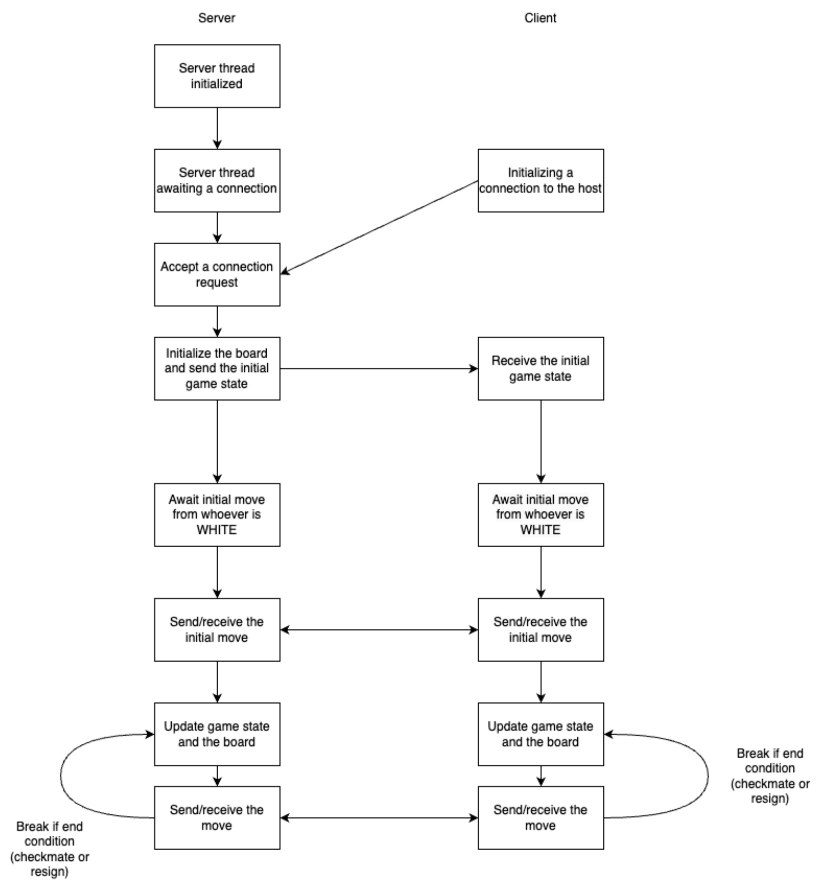
Determine what kind of information needs to be stored on each host, and what kind of information needs to be communicated between them. Hand in a design document, which describes your system's objective(s), issues that you anticipate, alternative solutions to those issues and explanation of your choice of solutions (e.g. application architecture, choice of transport-layer protocol, design issues such as synchronization and maintaining consistency of data), application-layer protocols (e.g. what is the format and order of messages that hosts send to eachother), algorithms, and any assumptions/limitations/restrictions of the system. The bolded items are the most important for this assignment -- it is recommended that you answer these in the form of a diagram, e.g. like on the whiteboard under labs 1 and 2 (updated as relevant for your application, with the red arrows representing exchange of data; label the arrows with what data is being exchanged). The diagram(s) will:

- **show how the minimalist version of your network application works; version 1 is also known as the rapid prototype.**
- **show how you plan to enhance your application, involving multiple clients, and issues of synchronization and consistency of data. How will the synchronization issue be handled? Explain in words, not code (yet). Optional: What happens if each client maintains a cache of a portion (e.g. one-tenth) of the information stored at the server? You may choose additional enhancements such as a GUI interface, but that is not required.**

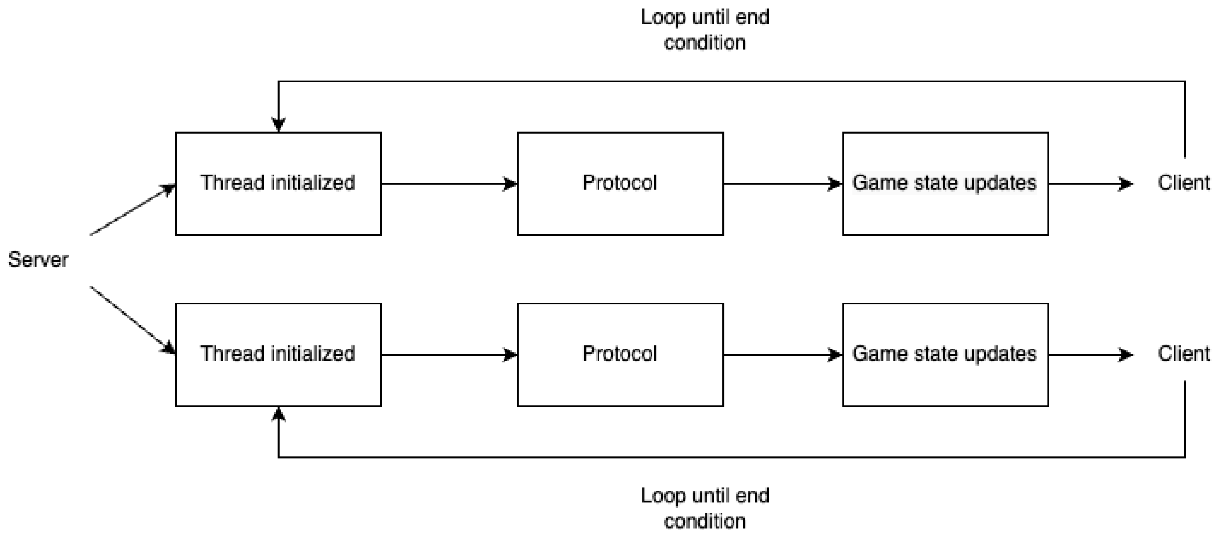
The minimalist version of the project will include a client playing against the server in a game of chess as shown in this diagram:


Server
Client
Server thread initialized
Server thread awaiting a connection
Initializing a connection to the host
Accept a connection request
Initialize the board and send the initial game state
Receive the initial game state
Await initial move from whoever is WHITE
Await initial move from whoever is WHITE
Send/receive the initial move
Send/receive the initial move
Update game state and the board
Update game state and the board
Break if end condition (checkmate or resign)
Send/receive the move
Send/receive the move
Break if end condition (checkmate or resign)

Client:
Stores the current game state (board configuration, player color, turn status).
Sends player moves to the server.
Receives updates from the server about the opponent's moves and game state changes.

Server:
Stores the complete game state (board configuration, player colors, turn status).
Validates player moves and updates the game state accordingly.
Sends updates to clients about the opponent's moves and game state changes.

The enhanced version of the project would include the support of multiple client concurrently playing at the same time with another client:



Each client will maintain the board and will receive information from a server thread with a move update (for example: moves = {e2e4}) where the first part e2 is what piece moves, and e4 is where it moves. Since it isn't viable to send the full board, what is going to be sent between the threads and the client is the json file with game state updates which would also include whose turn it is (white vs black). Since each client would maintain game state information, such as what color the client is playing as, that would solve the problem of having both clients making a move at the same time. The rules of the chess game will be followed through validation of a move using an api. An enhanced version should also be able to have a leaderboard. Every player would start with 1000 elo, and every win or loss is +-10 points.

Concurrency Control:

- Locks or other synchronization mechanisms to ensure mutual exclusion when accessing and modifying the game state on the server.
- Each client thread communicates with the server thread to request and receive updates on the game state.
- A shared object, such as a leaderboard, can be synchronized using locking mechanisms to prevent simultaneous modifications by multiple threads.

Leaderboard Synchronization:

- To synchronize the leaderboard, I will implement a locking mechanism where each thread must acquire a lock before modifying the leaderboard.
- When a client thread updates the leaderboard (ex: after a game outcome), it first acquires the lock, modifies the leaderboard data, and then releases the lock.
- This ensures that only one thread can modify the leaderboard at a time, preventing inconsistencies or data corruption.