Proposal to Fully Implement Networking and Database for OMG

Overview

Instead of using **stubs and drivers** to simulate server and database interactions, we are going to **fully implement the networking system and database integration**. This means that rather than calling placeholder functions that return fake data, the game will actually connect to a **server**, send and receive messages in real-time, and store game-related data in a **database**.

For the other teams, **this doesn't change how you work**—you'll still be calling the same types of functions, but now they will actually interact with a real system instead of a simulated one.

How the System Will Work at a High Level

System Structure

The system will be built using a **client-server architecture**. Here's how it will be structured:

1. **Client-Side (Game, GUI, and Networking Layer)**
    - The **Graphical User Interface (GUI)** will be responsible for rendering the game board, player menus, and match results.
    - The **game logic** will still run **on the client side**, but it will now **communicate with the server** for game state updates instead of assuming local control.
    - The **networking layer** will handle sending and receiving messages between the **game and the server**.
2. **Server-Side (Game Management & Database Layer)**
    - The **server** will act as the central authority, managing game sessions, handling player actions, and ensuring that all players are synchronized.
    - The **authentication system** (handling logins, registrations, and player sessions) will run on the server and interact with the **database** to validate users.
    - The **database** will store user accounts, match history, and leaderboards.

How Messages Will Be Sent and Processed

1. **When a Player Logs In:**

    - The **client sends login credentials** to the server.
    - The **server checks the database** to verify them.
    - If valid, the server **sends back a success message**, and the player is logged in.
2. **When a Player Joins a Game:**
    - The **client sends a request** to join a game.
    - The **server places them in a game session** and sends back game details.
    - The client **updates the GUI** based on the response.
3. **When a Player Makes a Move:**
    - The **client sends the move to the server**.
    - The **server verifies the move**, updates the game state, and sends the updated state back to all players.
    - Each client **updates their game board** to reflect the new state.
4. **When a Game Ends:**
    - The **server records the match results** in the database.
    - The **leaderboard is updated**, and players can view their past matches.

How This Affects Other Teams

Game Logic Team

- No changes to how you write the game logic.
- Instead of assuming that the game state is stored locally, you'll be sending and receiving updates from the **server**.
- The game logic still decides how moves work—**the server just ensures fairness and synchronization**.

UI/Frontend Team

- The UI will still call **the same functions**, but now they will actually send messages to the server and receive real responses.
- If you need **test responses**, you can still use placeholders, and we will integrate the real server connection later.

Other Teams

- If you want to **connect your work to the networking system yourself**, that's fine.
- If you'd rather keep using placeholders for now, **we will implement the real connections later**.

How We Are Going to Implement It

Step 1: Set Up the Database

- The **database will store user accounts, game results, and leaderboards**.
- Passwords will be **securely stored**, and only authenticated players will be able to access game sessions.

Step 2: Build the Server

- The **server will handle authentication, game sessions, and move validation**.
- It will be built to handle **multiple players at once** and make sure all players see the same game state.

Step 3: Connect the Game to the Server

- The **game will send and receive messages** instead of storing everything locally.
- Every action (logging in, making a move, winning a game) will involve **sending a request to the server and waiting for a response**.

Step 4: Store Game Data in the Database

- The **server will store completed game results** so that players can see past matches.
- The **leaderboard will be automatically updated** when a player wins or loses.

Step 5: Security & Testing

- Make sure the system can handle **multiple players at once**.
- Ensure **no one can cheat** by modifying their game state locally.
- Validate all incoming data to **prevent hacking attempts**.

Why We Should Do This Instead of Using Stubs

1. No Extra Work for Other Teams

- The **only** team doing additional work is **our networking team**—all other teams can continue working as usual.
- Instead of making **fake function calls**, other teams will be making **real function calls**—but their code will remain **exactly the same**.

2. Higher Grade Without More Work for Others

- Our professor encouraged us to **go above and beyond**—this is a way to do that **without disrupting other teams**.
- This ensures we **stand out** and have a chance at the **best possible grade** without requiring anyone else to change what they're doing.

3. We Can Put This on Our Resumes & Talk About It in Interviews

- Instead of just saying we worked on a project, we will be able to say we **built a fully functional online multiplayer game**.
- This kind of experience is **exactly what employers look for**—handling **networking, game state synchronization, and database integration** is something most projects don't include.
- It shows **real-world experience**, not just school-level assignments.

Final Thoughts & Next Steps

We will confirm with the professor on Tuesday if we can move forward with this. If approved:

- Other teams can **integrate their work whenever they want**—or they can keep using placeholders, and we will replace them later.
- We will **keep everyone updated** on our progress so integration is smooth.