1. Triqui o tres en raya

Para crear un **juego de Triqui (Tres en Raya)** usando una arquitectura **cliente-servidor con WebSocket o Threads**, es clave definir criterios técnicos, de diseño y funcionales que garanticen una experiencia fluida y segura. Aquí están los principales criterios y pasos:

1. **Arquitectura General**

- **Backend (Servidor):**
    - Gestiona la lógica del juego, conexiones WebSocket y sincronización entre clientes.
    - Tecnologías sugeridas: Node.js con ws o Socket.io, Python con FastAPI y WebSockets, o Java con Spring WebSocket.
- **Frontend (Cliente):**
    - Interfaz gráfica para los jugadores (HTML/CSS/JavaScript + React/Vue.js o frameworks móviles).
- **Protocolo de Comunicación:** WebSocket para comunicación bidireccional en tiempo real.

2. **Requisitos Funcionales**

Servidor:

- **Gestión de conexiones:**
    - Aceptar múltiples clientes y emparejarlos en partidas (1 vs 1).
    - Manejar desconexiones y reconexiones (ej.: notificar al otro jugador si alguien abandona).
- **Lógica del juego:**
    - Validar movimientos (turnos, posiciones válidas).
    - Detectar victorias/empates (patrones de 3 en línea).
    - Notificar a ambos clientes el estado actualizado del tablero.
    - El que primero gane 3 partidas de 5 es el ganador.
    - Se puede reiniciar el juego con el mismo jugador o en espera de otro jugador.
- **Sincronización:**
    - Mantener un estado único del juego en el servidor y replicarlo a los clientes.

Cliente:

- **Interfaz de usuario:**
    - Renderizar el tablero de 3x3.
    - Permitir hacer clic en casillas para enviar movimientos al servidor.

- Mostrar mensajes de estado (turno actual, ganador, empate).

- **Comunicación con el servidor:**
  - Enviar movimientos al servidor via WebSocket.
  - Recibir actualizaciones del tablero en tiempo real.

3. Criterios Técnicos Clave

Mensajes WebSocket (Estructura):

- Definir un formato estándar para los mensajes (ej.: JSON):

Gestión de Partidas en el Servidor:

- **Crear una clase/servicio** GameManager para:
  - Almacenar partidas activas en memoria o base de datos.
  - Asignar IDs únicos a cada partida.
  - Asignar símbolos (X/O) a los jugadores al emparejarlos.

Validación y Seguridad:

- **Evitar trampas:**
  - Asegurar que el servidor sea la única fuente de verdad (no confiar en el cliente).
  - Validar que los movimientos recibidos correspondan al turno correcto.

- **Proteger contra ataques:**
  - Sanitizar datos recibidos (ej.: posiciones dentro del rango 0-2).
  - Limitar la tasa de mensajes por cliente.

4. Flujo de Comunicación

5.

4.1. **Conexión inicial:**

Cliente se conecta al servidor WebSocket.
Servidor asigna al jugador a una partida (nueva o existente) y envía gameId y símbolo (X/O).

4.2. **Durante la partida:**

Cliente envía { "type": "move", "position": [1, 1] }.
Servidor valida el movimiento, actualiza el tablero, verifica si hay ganador.
Servidor envía a ambos clientes el nuevo estado del tablero y el turno actual.

4.3. **Finalización:**

Si hay ganador o empate, el servidor notifica a los clientes.
Opción de reiniciar la partida (type: "reset").

6. **Diseño de la Interfaz (Cliente)**
    - **Tablero visual:**
        - Usar CSS Grid o Canvas para una cuadrícula 3x3.
        - Mostrar X/O con iconos o textos estilizados.
    - **Feedback en tiempo real:**
        - Bloquear el tablero cuando no es el turno del jugador.
        - Animaciones para movimientos y victorias.
    - **Mensajes de estado:**
        - "Esperando jugador...", "Tu turno (X)", "¡Ganó O!".
7. **Manejo de Errores y Casos Extremos**
    - **Desconexiones:**
        - Si un jugador se desconecta, notificar al otro y ofrecer reiniciar.
    - **Tiempos de espera:**
        - Si un jugador no realiza un movimiento en X segundos, penalizar o finalizar la partida.
    - **Mensajes inválidos:**
        - Enviar { "type": "error", "message": "Movimiento no válido" }.
8. **Herramientas y Librerías Recomendadas**
    - **Backend:**
        - Node.js + ws (liviano) o Socket.io (con reconexión automática).
        - Python: FastAPI + websockets.
        - Java, .net
    - **Frontend:**
        - Vanilla JavaScript + WebSocket API o Socket.io-client.
        - React/Vue.js para componentes dinámicos.
    - **Testing:**
        - Postman (para probar WebSocket) o herramientas como Jest (backend).
9. **Extras Opcionales**
    - **Chat en tiempo real:** Usar canales WebSocket adicionales para mensajes entre jugadores.
    - **Leaderboards:** Almacenar victorias en una base de datos (ej.: MongoDB, PostgreSQL).
    - **Modo single-player vs IA:** Integrar un algoritmo minimax para jugar contra la máquina.

10. **Arquitectura de software:**

- **Aplicar principios SOLID**.
- **Aplicar patrones de diseño:** Tiene gran importancia en la nota, se deben tener claro en donde se aplican y cuál es la razón desde el punto de vista de diseño.