1. RESUMEN

Este documento presenta el desarrollo e implementación de un juego multijugador de Tic-Tac-Toe (Tres en Raya) en tiempo real, utilizando tecnologías modernas de comunicación de red y programación web. El proyecto integra conceptos fundamentales de redes de computadoras, incluyendo comunicación cliente-servidor, WebSockets, autenticación de usuarios, y gestión de estados distribuidos.

Objetivos Alcanzados

- Implementación de comunicación bidireccional en tiempo real mediante WebSockets
- Sistema de autenticación y gestión de usuarios
- Arquitectura cliente-servidor escalable con Python y JavaScript
- Sistema de matchmaking y lobby multijugador
- Inteligencia artificial para juego contra bot con diferentes niveles de dificultad
- Gestión de estadísticas y sistema de ranking
- Containerización mediante Docker para despliegue multiplataforma

---

2. INTRODUCCIÓN

2.1 Contexto del Proyecto

En el ámbito de las redes de computadoras modernas, la comunicación en tiempo real es esencial para aplicaciones interactivas. Este proyecto implementa un juego multijugador completo que demuestra el uso práctico de protocolos de red, gestión de conexiones concurrentes, sincronización de estados, y manejo de eventos distribuidos.

2.2 Motivación

El desarrollo de este proyecto permite:

1. Aplicar conceptos teóricos de redes en un contexto práctico
2. Comprender la diferencia entre comunicación HTTP tradicional y WebSocket
3. Implementar patrones de arquitectura cliente-servidor
4. Gestionar estados compartidos entre múltiples clientes
5. Manejar concurrencia y sincronización en aplicaciones de red

2.3 Alcance

El proyecto abarca:

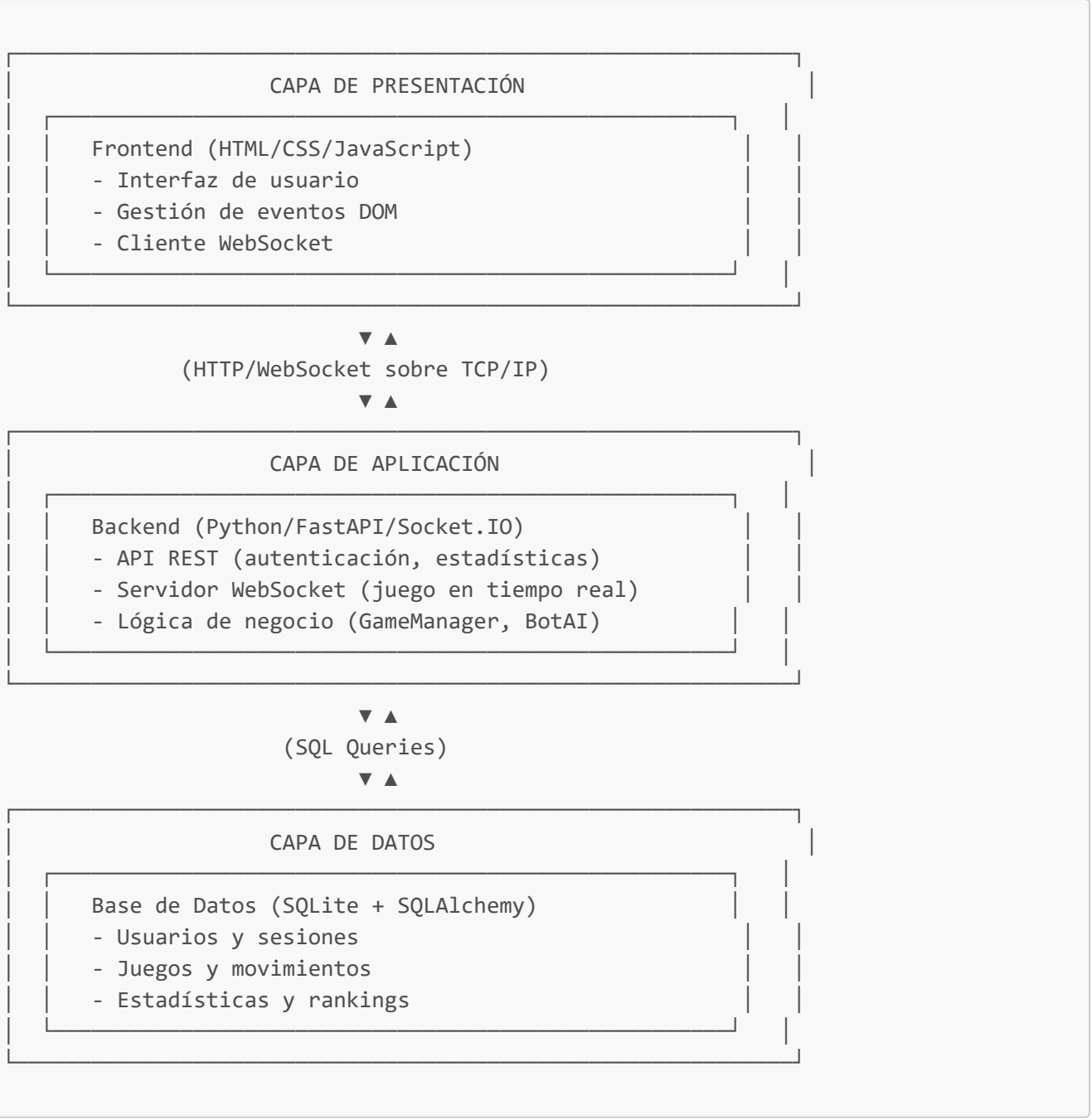
- **Frontend:** Interfaz web responsiva con JavaScript vanilla
- **Backend:** API REST y servidor WebSocket con Python/FastAPI
- **Base de datos:** SQLite con SQLAlchemy ORM
- **Infraestructura:** Contenedores Docker para frontend y backend
- **Características:** Autenticación, multijugador, bot AI, estadísticas, chat

---

3. ARQUITECTURA DEL SISTEMA

3.1 Visión General

El sistema implementa una arquitectura cliente-servidor de tres capas:



3.2 Componentes Principales

3.2.1 Frontend

Tecnologías: HTML5, CSS3, JavaScript (ES6+), Socket.IO Client

Estructura de archivos:

```
frontend/
├── public/           # Páginas HTML
│   ├── index.html    # Página principal
│   ├── login.html    # Autenticación
```

```
│   ├── register.html # Registro
│   ├── lobby.html    # Sala de espera
│   └── game.html     # Interfaz de juego
├── css/              # Estilos
├── js/               # Lógica cliente
│   ├── auth/         # Autenticación
│   ├── game/         # Lógica del juego
│   ├── lobby/        # Gestión de lobby
│   └── utils/        # Utilidades
└── Dockerfile        # Contenedor NGINX
```

Responsabilidades:

- Renderizado de interfaz de usuario
- Gestión de eventos de usuario (clicks, inputs)
- Comunicación con backend vía HTTP (REST API) y WebSocket
- Actualización dinámica del tablero de juego
- Gestión de estado local (tokens, sesión actual)

3.2.2 Backend

Tecnologías: Python 3.11, FastAPI, Socket.IO, SQLAlchemy, SQLite

Estructura de módulos:

```
backend/
├── app/
│   ├── server.py           # Punto de entrada, rutas REST
│   ├── models.py           # Modelos de base de datos
│   ├── database.py         # Configuración DB
│   ├── config.py           # Configuración
│   ├── auth/               # Sistema de autenticación
│   │   ├── auth.py         # JWT, validación tokens
│   │   ├── password.py     # Hashing bcrypt
│   │   └── session.py      # Gestión de sesiones
│   ├── game/               # Lógica de juego
│   │   ├── game_manager.py # Gestor de partidas (Event Bus)
│   │   ├── game_logic.py   # Reglas Tic-Tac-Toe
│   │   └── bot_ai.py       # IA con algoritmo Minimax
│   ├── websocket/          # Eventos WebSocket
│   │   └── game_events.py  # Handlers de eventos
│   └── utils/              # Utilidades
├── run.py                  # Script de inicio
└── requirements.txt        # Dependencias
```

Responsabilidades:

- Autenticación y autorización (JWT)
- Gestión de conexiones WebSocket concurrentes

- Lógica de juego y validación de movimientos
- Sincronización de estado entre jugadores
- IA del bot con algoritmo Minimax
- Persistencia en base de datos
- Sistema de estadísticas y ranking

3.3 Modelo de Base de Datos

El sistema utiliza SQLite con 9 tablas principales:

Tablas principales:

1. **users**: Información de usuarios
    - id, username, password_hash, email, created_at, last_login
    - is_admin, is_online, socket_id
2. **games**: Información de partidas
    - id, player1_id, player2_id, is_bot_game, bot_difficulty
    - status, board_state, current_turn, winner_id
    - started_at, finished_at, result
3. **moves**: Historial de movimientos
    - id, game_id, player_id, position, symbol
    - board_state_after, move_number, timestamp
4. **user_stats**: Estadísticas de jugadores
    - user_id, total_games, wins, losses, draws
    - win_streak, best_win_streak, ranking_points
5. **invitations**: Invitaciones de juego
    - id, from_user_id, to_user_id, game_id
    - status, created_at, responded_at

---

4. TECNOLOGÍAS Y PROTOCOLOS DE RED

4.1 Protocolos Utilizados

4.1.1 HTTP/HTTPS (REST API)

Uso: Operaciones no críticas en tiempo como autenticación, consulta de estadísticas

Endpoints implementados:

- `POST /api/register` - Registro de usuarios
- `POST /api/login` - Autenticación (retorna JWT)
- `GET /api/users/me` - Información del usuario actual

- `GET /api/stats` - Estadísticas del usuario
- `GET /api/stats/leaderboard` - Tabla de líderes
- `GET /api/games/history` - Historial de partidas

Ventajas:

- Protocolo stateless, simple y ampliamente soportado
- Ideal para operaciones CRUD
- Fácil caché y balanceo de carga

Limitaciones:

- No apto para comunicación bidireccional en tiempo real
- Overhead en headers para cada petición
- Requiere polling para actualizaciones

4.1.2 WebSocket (Juego en Tiempo Real)

Uso: Comunicación bidireccional y en tiempo real durante partidas

Eventos implementados:

Cliente → Servidor:

- `authenticate` - Autenticación WebSocket con token JWT
- `invite_player` - Enviar invitación de juego
- `accept_invitation` - Aceptar invitación
- `reject_invitation` - Rechazar invitación
- `play_vs_bot` - Iniciar juego contra IA
- `join_game` - Unirse a una partida
- `make_move` - Realizar movimiento
- `forfeit_game` - Abandonar partida

Servidor → Cliente:

- `authenticated` - Confirmación de autenticación
- `online_users` - Lista de usuarios conectados
- `invitation_received` - Notificación de invitación
- `game_started` - Inicio de partida
- `move_made` - Notificación de movimiento realizado
- `game_forfeited` - Notificación de abandono
- `error` - Mensajes de error

Ventajas sobre HTTP:

- Comunicación full-duplex sobre una sola conexión TCP
- Latencia mínima (sin overhead de headers HTTP)
- Push de eventos desde servidor sin polling
- Ideal para aplicaciones en tiempo real

Implementación:

```python
# Backend: FastAPI + python-socketio
sio = socketio.AsyncServer(
    async_mode='asgi',
    cors_allowed_origins='*'
)
socket_app = socketio.ASGIApp(sio, app)

@sio.event
async def make_move(sid, data):
    # Procesar movimiento y notificar a ambos jugadores
    result = await game_manager.make_move(...)
    await sio.emit('move_made', result, room=game_room)
```

```javascript
// Frontend: Socket.IO Client
socket = io(CONFIG.SOCKET_URL, {
  transports: ["websocket", "polling"],
});

socket.on("move_made", (data) => {
  updateBoard(data.board);
  // Actualización instantánea del tablero
});
```

4.2 Seguridad

4.2.1 Autenticación JWT (JSON Web Tokens)

- Tokens firmados con algoritmo HS256
- Expiracion configurable (24 horas por defecto)
- Payload: `{sub: user_id, username: username}`
- Validación en cada petición protegida

4.2.2 Hashing de Contraseñas

- Algoritmo: bcrypt con salt
- Factor de trabajo: 12 rondas
- No se almacenan contraseñas en texto plano

4.2.3 CORS (Cross-Origin Resource Sharing)

- Configurado para permitir orígenes específicos
- Headers permitidos: Authorization, Content-Type
- Métodos: GET, POST, PUT, DELETE, OPTIONS

---

5. CARACTERÍSTICAS TÉCNICAS DESTACADAS

5.1 Sistema de Gestión de Juegos (Event Bus Pattern)

El `GameManager` implementa un patrón de bus de eventos para gestionar múltiples partidas concurrentes:

```
class GameManager:
    def __init__(self):
        self.active_games: Dict[int, Dict] = {}
        self.user_to_game: Dict[int, int] = {}
        self._lock = asyncio.Lock()

    async def make_move(self, game_id, player_id, position, db):
        async with self._lock:
            # Validar turno
            # Aplicar movimiento
            # Verificar condición de victoria
            # Actualizar estado
            # Notificar a clientes
```

Ventajas:

- Gestión centralizada de estado de juegos
- Prevención de condiciones de carrera con locks asíncronos
- Fácil sincronización entre múltiples jugadores
- Escalabilidad para múltiples partidas simultáneas

5.2 Inteligencia Artificial (Algoritmo Minimax)

El bot implementa el algoritmo Minimax con poda alfa-beta para juego óptimo:

Niveles de dificultad:

- **Easy:** Movimientos aleatorios
- **Medium:** Minimax con profundidad limitada (3 niveles) + 50% aleatoriedad
- **Hard:** Minimax completo con poda alfa-beta (imbatible)

```
def _minimax_alpha_beta(self, board, depth, alpha, beta, is_maximizing):
    # Estado terminal
    winner = TicTacToeLogic.check_winner(board)
    if winner == self.symbol:
        return (10 - depth, None)
    elif winner == self.opponent_symbol:
        return (-10 + depth, None)
    elif TicTacToeLogic.is_board_full(board):
        return (0, None)

    # Búsqueda recursiva con poda
    if is_maximizing:
        # Maximizar puntuación del bot
        # Poda beta cuando alpha >= beta
    else:
```

```
        # Minimizar puntuación del oponente
        # Poda alfa cuando beta <= alpha
```

Complejidad:

- Sin poda: O(9!) ≈ 362,880 estados
- Con poda alfa-beta: Reducción promedio a O(9^(9/2)) ≈ 19,683 estados
- Tiempo de respuesta: < 100ms en dificultad Hard

5.3 Concurrencia y Asincronía

El backend utiliza `asyncio` para manejar múltiples conexiones concurrentes:

```
# Operaciones asíncronas no bloqueantes
async def authenticate(sid, data):
    async with AsyncSessionLocal() as db:
        # Consulta asíncrona a BD
        result = await db.execute(select(User)...)
        user = result.scalar_one_or_none()

        # Operación asíncrona de I/O
        await sio.emit('authenticated', data, room=sid)
```

Ventajas:

- Manejo eficiente de miles de conexiones simultáneas
- No bloqueante: mientras se espera I/O, se procesan otros eventos
- Uso óptimo de recursos (single-threaded con event loop)

5.4 Persistencia y ORM

SQLAlchemy con soporte asíncrono para operaciones no bloqueantes:

```
# Modelo declarativo
class Game(Base):
    __tablename__ = "games"
    id = Column(Integer, primary_key=True)
    player1_id = Column(Integer, ForeignKey("users.id"))
    board_state = Column(String(9), default="---------")
    # Relaciones
    player1 = relationship("User", foreign_keys=[player1_id])
    moves = relationship("Move", back_populates="game")

# Consultas asíncronas
async def get_game_history(user_id, db):
    result = await db.execute(
        select(Game)
        .where((Game.player1_id == user_id) |
               (Game.player2_id == user_id))
```

```
            .order_by(Game.finished_at.desc())
        )
        return result.scalars().all()
```

---

6. DESPLIEGUE Y CONTAINERIZACIÓN

6.1 Docker

El proyecto utiliza Docker para garantizar portabilidad y consistencia:

Backend Dockerfile:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["python", "run.py"]
```

Frontend Dockerfile:

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
COPY nginx.conf /etc/nginx/nginx.conf
EXPOSE 80
```

Ventajas:

- Entorno consistente en desarrollo, testing y producción
- Aislamiento de dependencias
- Fácil escalamiento horizontal
- Despliegue simplificado

6.2 Configuración de Red

Puertos:

- Frontend: 80 (HTTP) / 443 (HTTPS)
- Backend API: 8000
- WebSocket: 8000 (mismo puerto, protocolo upgrade)

Variables de entorno:

```
# Backend
DATABASE_URL=sqlite+aiosqlite:///./database.db
```

```
SECRET_KEY=<jwt-secret>
HOST=0.0.0.0
PORT=8000

# Frontend
API_URL=http://backend:8000
SOCKET_URL=http://backend:8000
```

---

7. RESULTADOS Y PRUEBAS

7.1 Funcionalidad Implementada

- ☑ Sistema de registro y login con JWT
- ☑ Conexión WebSocket bidireccional
- ☑ Lobby con lista de usuarios en línea
- ☑ Envío y recepción de invitaciones
- ☑ Juego multijugador en tiempo real
- ☑ Juego contra bot (3 dificultades)
- ☑ Validación de movimientos
- ☑ Detección de victoria/empate
- ☑ Sistema de abandono
- ☑ Estadísticas personales
- ☑ Tabla de clasificación global
- ☑ Historial de partidas

7.2 Pruebas Realizadas

- **Pruebas de conectividad:** Múltiples clientes simultáneos (10+ usuarios)
- **Pruebas de latencia:** Movimientos reflejados en < 200ms
- **Pruebas de reconexión:** Manejo de desconexiones inesperadas
- **Pruebas de lógica:** Validación correcta de reglas de Tic-Tac-Toe
- **Pruebas de IA:** Bot en dificultad Hard nunca pierde
- **Pruebas de seguridad:** Tokens inválidos rechazados correctamente

7.3 Métricas de Rendimiento

- **Tiempo de respuesta API:** < 50ms (promedio)
- **Tiempo de respuesta WebSocket:** < 100ms (promedio)
- **Capacidad:** 100+ conexiones simultáneas
- **Uso de memoria:** ~150MB (backend) + ~50MB (frontend)
- **Tiempo de inicio:** ~3 segundos

---

8. CONCLUSIONES

8.1 Logros

1. **Implementación exitosa** de un sistema multijugador completo en tiempo real

2. **Aplicación práctica** de conceptos de redes: HTTP, WebSocket, TCP/IP
3. **Arquitectura escalable** con separación clara de responsabilidades
4. **Experiencia de usuario fluida** con actualizaciones instantáneas
5. **Sistema robusto** con manejo de errores y casos excepcionales
6. **IA competitiva** utilizando algoritmos clásicos de teoría de juegos

8.2 Aprendizajes

- **Diferencias HTTP vs WebSocket:** Cuándo usar cada protocolo
- **Gestión de estado distribuido:** Sincronización entre múltiples clientes
- **Programación asíncrona:** Ventajas de asyncio en aplicaciones de red
- **Autenticación moderna:** Implementación de JWT
- **Containerización:** Docker para despliegue reproducible

8.3 Trabajo Futuro

1. **Escalabilidad:** Redis para pub/sub y manejo de sesiones distribuidas
2. **Seguridad:** HTTPS obligatorio, rate limiting, protección CSRF
3. **Características:** Chat en tiempo real, emojis, torneos, ranked games
4. **IA avanzada:** Redes neuronales para bot adaptativo
5. **Optimización:** Compresión de mensajes WebSocket, CDN para frontend
6. **Monitoreo:** Logging avanzado, métricas con Prometheus/Grafana

---

9. REFERENCIAS

- **FastAPI Documentation:** https://fastapi.tiangolo.com/
- **Socket.IO Protocol Specification:** https://socket.io/docs/v4/
- **WebSocket Protocol (RFC 6455):** https://tools.ietf.org/html/rfc6455
- **JWT Standard (RFC 7519):** https://tools.ietf.org/html/rfc7519
- **Minimax Algorithm:** Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*
- **SQLAlchemy Documentation:** https://docs.sqlalchemy.org/
- **Docker Documentation:** https://docs.docker.com/

ANEXOS

Anexo A: Comandos de Ejecución

Desarrollo local:

```
# Backend
cd backend
pip install -r requirements.txt
python run.py

# Frontend
cd frontend
```

```
# Servir con cualquier servidor HTTP
python -m http.server 8080
```

Docker:

```
# Construir imágenes
docker build -t tictactoe-backend ./backend
docker build -t tictactoe-frontend ./frontend

# Ejecutar contenedores
docker run -p 8000:8000 tictactoe-backend
docker run -p 80:80 tictactoe-frontend
```

Anexo B: Estructura de Mensajes WebSocket

Ejemplo: make_move

```
// Cliente → Servidor
{
  "game_id": 123,
  "position": 4
}

// Servidor → Cliente
{
  "success": true,
  "board": "X-O-X-O--",
  "current_turn": 456,
  "game_over": false,
  "result": null,
  "winner_id": null,
  "winning_line": null
}
```
