Sección 1: Diseño Técnico y Arquitectural

Problema: Diseñe un sistema SENCILLO escalable y distribuido para un juego multijugador en línea en tiempo real.

Considere factores como:

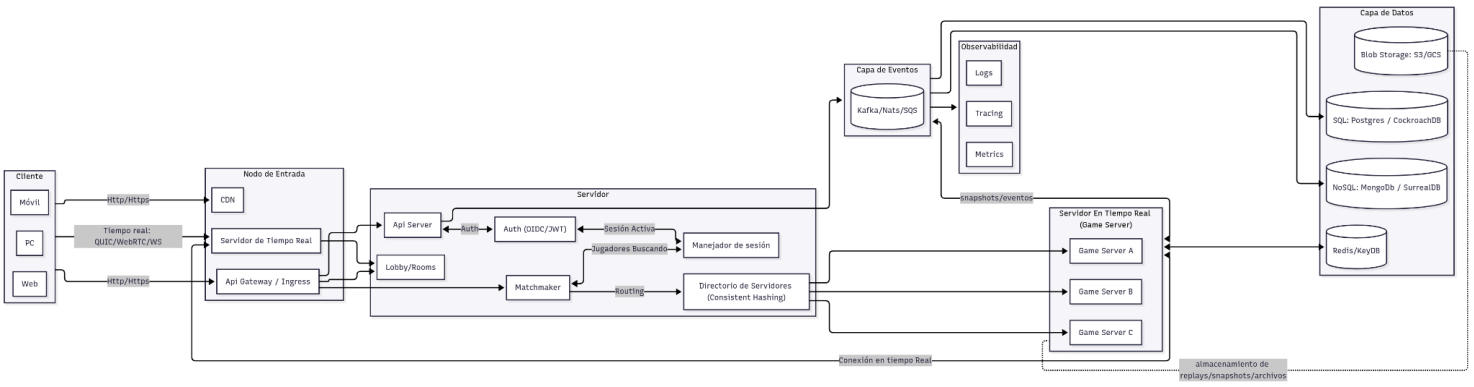
- Alta concurrencia y baja latencia.
- Consistencia y fiabilidad de los datos.
- Escalabilidad y tolerancia a fallos.
- Seguridad y privacidad.

Entregables:

- Diagrama arquitectónico: Una representación visual clara y concisa de los componentes, interacciones y flujo de datos del sistema.
- Documento de diseño. Esto significa:
    - Stack tecnológico y justificación.
    - Estrategias de almacenamiento y gestión de datos.
    - Mecanismos de escalabilidad.
    - Tolerancia a fallos.
    - Medidas de seguridad.

Solución

Diagrama Arquitectónico



Si bien los juegos multijugador requieren conexiones en tiempo real para mantener la interacción entre los jugadores, no todos los componentes del sistema necesitan operar bajo la misma exigencia. Lo que quiere decir que existen funciones críticas que sí dependen de la latencia mínima (como la sincronización de posiciones, disparos o colisiones en pantalla), mientras que otras pueden manejarse de forma asíncrona o en lotes (por ejemplo, el registro de métricas, el almacenamiento de historiales de partidas o la actualización de inventarios).

Diseño

2.1) Stack Tecnologico

Lenguajes:

Para la capa de Game Server se recomienda el uso de lenguajes de alto rendimiento como C++, Go, Rust o Zig (y si se busca un enfoque extremo, incluso C). La razón es que estos ofrecen rendimiento más o menos predecible, baja latencia y control sobre recursos de memoria y concurrencia, lo cual es esencial para procesar en tiempo real la lógica del juego (física, colisiones, sincronización de estado).

Herramientas de construcción de juegos

Tenemos infinidades de herramientas que nos sirven para esto, e incluso en los lenguajes ya antes mencionados, como por ejemplo: Unity, Unreal Engine, Godot, Gamemaker, etc.

Transporte en Tiempo Real:

- **WebRTC DataChannels (sobre QUIC):** óptimo para clientes en navegador, dado que ofrece baja latencia y NAT traversal integrado.
- **WebSockets:** funcionan como fallback universal, ya que son ampliamente soportados y fáciles de implementar en cualquier cliente.
- **UDP/QUIC:** para clientes que lo soporten, proporciona máxima eficiencia al evitar la sobrecarga de TCP, permitiendo enviar/recibir estados del juego en milisegundos.

También sería posible combinar los diferentes tipos de transporte dependiendo del caso de uso.

Nodos de Entrada:

Podríamos entregar contenidos estaticos utilizando **CDNs**, y también podrían protegernos de ataques de DDos. Además adicionar un **Loadbalancer/GlobalLoadBalancer** que reduciría la latencia llevando a los jugadores a los servidores más cercanos a ellos. Adicional tener en cuenta también una **ApiGateway** para distribuir las cargas entre las diferentes instancias de los servidores.

Orquestación:

Sugeriría Kubernetes o alguna solución similar, para:

- Escalar horizontalmente en múltiples regiones.
- Tolerar fallos automáticamente.
- Gestionar ciclos de vida de microservicios y servidores de juego de manera declarativa.

Esto podría hacer que nuestros servidores soporten cambios y picos de tráficos inesperados, sin necesidad de comprometer la disponibilidad.

Datos:

- **Redis / KeyDB:** para **sesiones efímeras y caché de baja latencia**, críticos en autenticación y sincronización rápida.
- **NATS / Kafka / Redis Streams:** como **event bus distribuido**, garantizando la entrega de mensajes entre servicios con resiliencia y escalabilidad.
- **SQL (Postgres / CockroachDB):** bases de datos con **consistencia fuerte** para datos persistentes (progresos, inventarios, transacciones económicas dentro del juego).
- **NoSQL(SurrealDB, DynamoDB, MongoDB):** Para datos que no necesitan una relación y los datos necesitan flexibilidad para almacenarse.
- **S3 u object storage equivalente:** almacenamiento de bajo costo para assets estáticos, replays, snapshots o backups fríos.

Observabilidad:

- **OpenTelemetry:** para estandarizar métricas y trazas en todos los servicios.
- **Prometheus:** para monitoreo de métricas en tiempo real.
- **Loki / ELK Stack:** para la centralización y análisis de logs.
- **Grafana:** como capa de visualización unificada, permitiendo a los equipos detectar problemas y responder con rapidez.

2.2) Estrategias de almacenamiento y gestión de datos

Datos que constantemente se están utilizando en el **Servidor de Juegos (Tiempo Real)** deben mantenerse en memoria del propio servidor o en un servicio de *in-memory cache* como **Redis/KeyDB**, lo que garantiza latencia mínima para operaciones críticas (movimiento, HP, colisiones).

Cuando surge la necesidad de **persistir información generada durante la partida** (ej. eventos, chat, métricas), estos datos pueden publicarse en el **bus de eventos** (NATS/Kafka/Redis Streams) y posteriormente almacenarse en una base **NoSQL**, lo cual ofrece escalabilidad y flexibilidad de esquema para manejar grandes volúmenes de eventos en formato *append-only*.

La **observabilidad (logging y tracing)** también se envía primero al bus para desacoplar la ingesta del almacenamiento, y desde allí a un sistema de persistencia NoSQL especializado (por ejemplo, Elastic o Loki) que permita búsquedas y análisis masivos en tiempo real.

Ahora bien, hay casos donde **SQL es la elección correcta**. Datos canónicos como **perfiles de jugador, progreso.** Porque sería mucho más sencillo extraer la información si está relacionada.

2.3) Mecanismos de escalabilidad.

Una de las maneras más efectivas de escalar el juego es a través de un Matchmaker elástico que asigne jugadores a distintas salas(Servidores de juego) de manera dinámica, evitando concentrar toda la carga en un único servidor. De esta forma, el sistema distribuye el tráfico de manera uniforme y garantiza que cada sala se mantenga dentro de parámetros saludables.

A su vez, los Game Servers se escalan horizontalmente en función de la cantidad de salas activas y de métricas clave como los percentiles P95/P99 de latencia, lo que permite reaccionar de forma automática a picos de demanda.

Como ya lo veníamos mencionando podríamos utilizar un sistema distribuido que se encargue de coordinar las cargas y balancear, ejm: Kubernets.

Para mantener la estabilidad, se aplican mecanismos de backpressure como límites de jugadores por sala, un *tick rate* configurable que balancea precisión con rendimiento, y compresión de requests para reducir el ancho de banda. Finalmente, se contempla la posibilidad de realizar migraciones de salas en caliente mediante snapshots del estado y redirección transparente de jugadores, asegurando continuidad incluso en escenarios de mantenimiento o redistribución de carga.

2.4) Tolerancia a fallos.

Podemos transmitir **heartbeats** desde los servidores, de esa manera podríamos darnos cuenta rápidamente la caída de un Game Server y activar la reasignación automática de jugadores hacia otra instancia disponible.

Para los jugadores podríamos implementar un **reconnect token,** con ello puedan almacenarlo de manera local, y cuando vuelva su conexión poder acceder de nuevo al gameserver al que estaban asignados.

Como ya venimos hablando, hay datos que podríamos enviar a la cola y no frenar nuestro procesamiento, por lo que podríamos implementar una persistencia híbrida (memoria + cola) antes de confirmar acciones críticas al jugador, se escriben también en el bus o en una DB fuerte, de modo que si el servidor muere, la acción ya está registrada.

2.5) Medidas de seguridad.

La comunicación entre cliente y servidor se podría asegurar con **TLS 1.3**, que cifra todo el tráfico, y en clientes modernos puede aprovechar **QUIC** (el protocolo base de HTTP/3) para combinar cifrado + baja latencia + resistencia a pérdidas de paquetes.

Para la **identidad de los jugadores**, podríamos utilizar **JWT (JSON Web Tokens)** firmados. Con estos tokens contendremos la información mínima de autenticación y expirará rápido, lo que los hace apropiados para sesiones efímeras. En caso de desconexión, el cliente puede enviar un **token de reconexión temporal** para reanudar la partida sin pasar por un login completo, siempre que no haya caducado ni haya sido revocado.

Y por último un AntiCheat, que se basa en un modelo de **servidor autoritativo**, lo que significa que el servidor nunca confía ciegamente en lo que envía el cliente. Todos los inputs del jugador (movimientos, disparos, acciones) son validados en el servidor antes de aplicarse a la lógica del juego. Esto impide trampas comunes como “speed hacks” o manipulación de memoria local. Además, se pueden añadir validaciones estadísticas (detección de patrones imposibles de reacción) o comprobaciones de integridad en el cliente, pero la capa crítica está siempre en el lado del servidor.

Conclusión

En conclusión, el diseño propuesto trata de encontrar un equilibrio entre simplicidad y robustez, garantizando que el sistema pueda escalar de manera horizontal y responder a la alta concurrencia propia de un juego multijugador en tiempo real. Separar lo que requiere latencia mínima de lo que puede procesarse de forma asíncrona permite optimizar recursos sin sacrificar la experiencia del jugador. La combinación de un stack tecnológico moderno, estrategias diferenciadas de almacenamiento, mecanismos elásticos de escalabilidad, planes de tolerancia a fallos y medidas de seguridad integradas, asegura un ecosistema flexible y resiliente. Con estas decisiones, se logra una arquitectura capaz de mantener la fluidez de las partidas, la coherencia de los datos críticos y la protección frente a ataques o trampas, sentando así las bases de un sistema confiable, escalable y seguro.