Informe de Diseño del Sistema: Agenda Distribuida

Integrantes: [Nombres de los Integrantes]

November 28, 2025

Abstract

El presente informe detalla el diseño y la arquitectura de un sistema de agenda distribuida, concebido para cumplir con los requisitos de alta disponibilidad y tolerancia a fallos en un entorno de múltiples nodos. Partiendo de una implementación centralizada basada en microservicios, se propone una evolución hacia un sistema robusto y escalable horizontalmente. La arquitectura se fundamenta en la replicación de servicios sin estado y la gestión consistente de servicios con estado mediante el algoritmo de consenso Raft. Se abordan los desafíos inherentes a los sistemas distribuidos, incluyendo la comunicación entre procesos, la coordinación, el descubrimiento de servicios, la consistencia de datos y la seguridad, con el objetivo de lograr una solución que sea indiferente para el usuario final frente a una implementación centralizada.

1 Introducción

La gestión eficiente del tiempo y la coordinación de actividades en grupo son problemas fundamentales en la interacción humana y profesional. El proyecto de Agenda Distribuida se presenta como una solución tecnológica a este desafío, permitiendo a los usuarios gestionar eventos personales y coordinar citas grupales de manera dinámica. Este sistema ha sido diseñado como una arquitectura de microservicios distribuidos, implementado en Go, donde la comunicación se gestiona a través de una combinación de API REST y un bus de mensajes Redis. El objetivo principal de este informe es detallar el diseño arquitectónico propuesto para un sistema distribuido plenamente funcional, capaz de operar sobre una infraestructura de red virtualizada en múltiples máquinas, cumpliendo con los requisitos académicos establecidos y resolviendo los problemas canónicos de la computación distribuida.

2 Diseño del Sistema Distribuido

2.1 Arquitectura

La organización del sistema se basa en un paradigma de microservicios robusto, diseñado para operar eficazmente en un entorno distribuido. Se definen cuatro roles principales: el **API Gateway Service**, que funciona como punto de entrada unificado y balanceador de carga; el **User Service** y el **Group Service**, concebidos como servicios sin estado (stateless) que encapsulan la lógica de negocio; y el **DB Service**, un servicio con estado

(stateful) cuya responsabilidad principal es asegurar la persistencia y consistencia de los datos.

Para la distribución de estos servicios sobre una red Docker Swarm, implementada en un mínimo de dos máquinas físicas, se contempla el despliegue de múltiples réplicas de los servicios sin estado (API Gateway, User, Group), que se asignarán de manera equitativa entre los nodos físicos disponibles. Esta estrategia no solo garantiza una alta disponibilidad sino que también facilita un balanceo de carga eficiente para las peticiones entrantes. En contraste, el DB Service se establecerá como un clúster de cinco nodos, distribuidos estratégicamente entre las máquinas físicas, que formarán un grupo de replicación coordinado por el algoritmo de consenso Raft. Este enfoque asegura la consistencia y la máxima tolerancia a fallos del estado crítico del sistema.

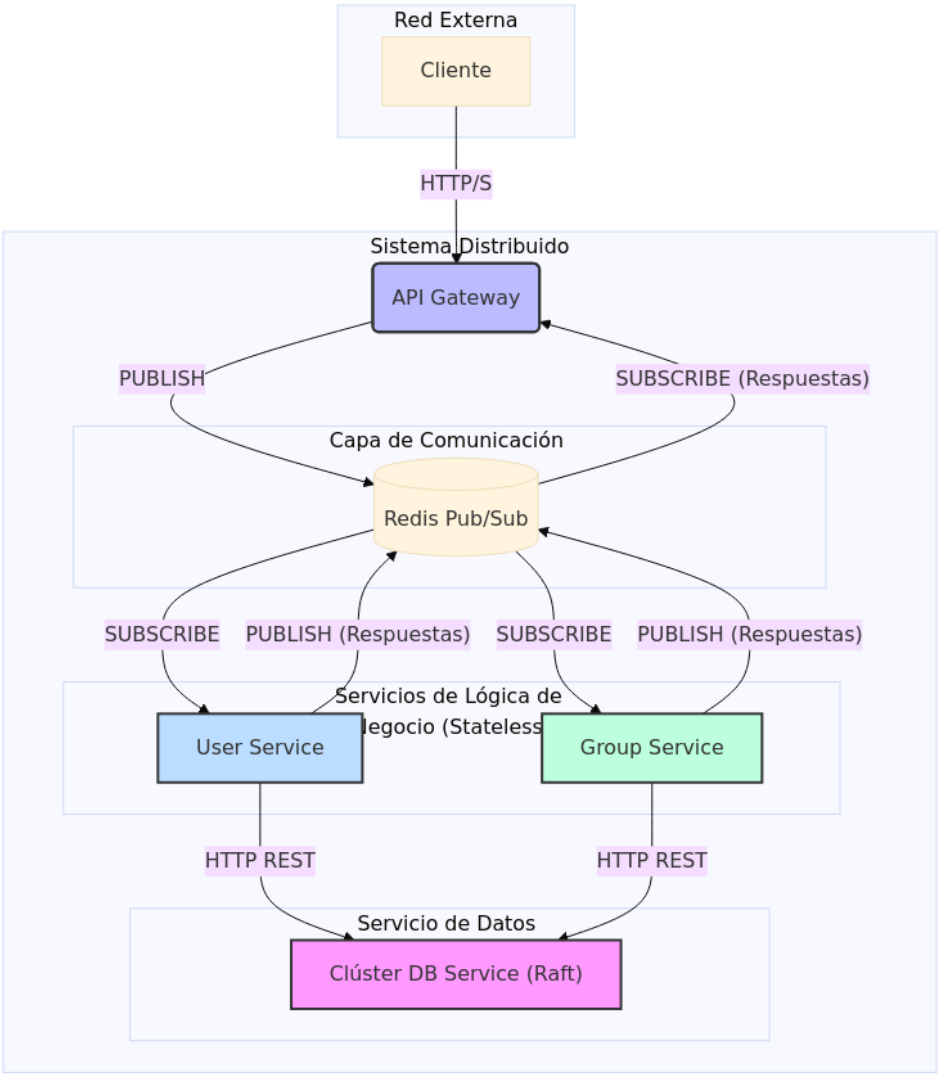

Figure 1: Diagrama General de la Arquitectura Distribuida

2.2 Procesos

El sistema se compone de procesos que encapsulan la funcionalidad de cada servicio. Cada instancia de un microservicio se ejecuta dentro de su propio contenedor Docker, lo

que garantiza el aislamiento y la portabilidad. Los procesos del User Service y Group Service son de naturaleza stateless; no almacenan datos críticos de sesión localmente, lo que permite que cualquier réplica pueda procesar cualquier petición de usuario. Esto simplifica la escalabilidad y la recuperación ante fallos, ya que una nueva instancia puede ser levantada para reemplazar a una caída sin pérdida de información.

El desempeño de cada proceso se optimiza mediante el uso del modelo de concurrencia nativo de Go. Las goroutines y los canales se emplean para gestionar múltiples peticiones de manera asíncrona y eficiente, permitiendo que cada instancia de servicio maneje una alta carga de trabajo sin bloquearse. Este patrón de diseño concurrente es fundamental para la capacidad de respuesta del sistema.

2.3 Comunicación

La comunicación en el sistema adopta un enfoque híbrido. La interacción entre el cliente final y el sistema se realiza a través de una API HTTP REST expuesta por el API Gateway. Este, a su vez, se comunica con los servicios internos de lógica de negocio (User y Group Service) utilizando el mismo protocolo.

La comunicación servidor-servidor, sin embargo, requiere mecanismos más sofisticados. Para la comunicación síncrona, como las consultas de los servicios de negocio al DB Service, se emplean llamadas HTTP REST dirigidas al nodo líder del clúster Raft. La comunicación asíncrona, utilizada para la notificación de eventos y el desacoplamiento entre servicios, se mantiene a través de patrones de mensajes (Publicación/Suscripción) sobre un bus de mensajes Redis. Finalmente, la comunicación entre los procesos que forman parte de un mismo clúster distribuido, como los nodos del DB Service (Raft) o el mecanismo de descubrimiento de servicios, se realiza a bajo nivel mediante sockets TCP/IP directos para maximizar la eficiencia.

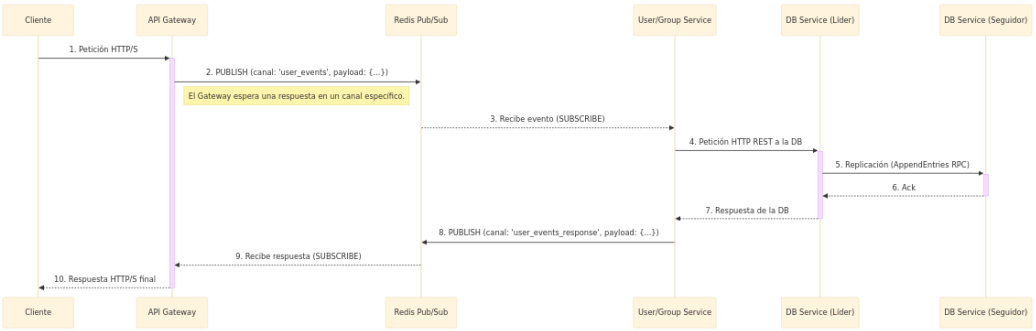

Figure 2: Diagrama de Secuencia de Comunicación Asíncrona

2.4 Coordinación

La coordinación de acciones y el acceso a recursos compartidos son los desafíos más críticos. El principal recurso compartido es la base de datos, y para gestionar el acceso concurrente a la misma se implementa el algoritmo de consenso Raft en el DB Service. Raft asegura que todas las operaciones de escritura sean linealizables al ser procesadas secuencialmente por un único líder elegido democráticamente. Cualquier modificación en los datos es replicada en un registro (log) a una mayoría de los nodos antes de ser

considerada como comprometida, previniendo así condiciones de carrera y garantizando la consistencia del estado.

Esta toma de decisiones distribuidas es el pilar de la consistencia del sistema. La elección de un líder y la replicación del log son procesos coordinados que permiten al clúster de base de datos actuar como una única máquina de estados lógica y tolerante a fallos.

2.5 Nombrado y Localización

En un entorno dinámico donde las instancias de servicios pueden ser creadas o destruidas, la localización de recursos es un problema no trivial. Depender del DNS de Docker no es suficiente y no cumple con los requisitos de robustez del proyecto. Para resolverlo, se propone la implementación de un mecanismo de descubrimiento de servicios descentralizado basado en un protocolo de Gossip.

Cada instancia de servicio, al iniciarse, se unirá a una red de "chismorreo" y periódicamente intercambiará información con otros nodos sobre los servicios que conoce y sus direcciones de red (IP y puerto). De esta manera, cada nodo construye y mantiene una vista local y actualizada de la topología del sistema. Cuando el API Gateway necesite enviar una petición a una réplica del User Service, por ejemplo, consultará su tabla local para obtener una lista de instancias saludables y seleccionará una, implementando así un balanceo de carga básico. Este enfoque elimina los puntos únicos de fallo asociados a los registros de servicio centralizados.

2.6 Consistencia y Replicación

La estrategia de consistencia y replicación varía según el tipo de servicio. Para los servicios sin estado, la replicación activa-activa es la norma. Se ejecutan múltiples réplicas idénticas para disponibilidad y balanceo de carga, pero no replican estado entre ellas.

Para el DB Service, que es stateful, la replicación es el núcleo de su diseño. Se utilizará un clúster de tres o cinco nodos, donde los datos se replican en cada uno de ellos. El modelo de consistencia garantizado por Raft es la consistencia fuerte (strong consistency). Una operación de escritura solo es confirmada al cliente una vez que ha sido replicada de forma segura en la mayoría de los nodos del clúster. Esto asegura que cualquier lectura posterior a una escritura confirmada devolverá el valor actualizado, previniendo lecturas de datos obsoletos, incluso ante fallos de red o de nodos individuales.

2.7 Tolerancia a Fallas

El sistema está diseñado para alcanzar un nivel de tolerancia a fallos de al menos 2. Esto significa que debe poder soportar la falla de al menos un nodo sin perder disponibilidad ni consistencia de los datos. Para los servicios sin estado, la caída de una réplica es gestionada por el API Gateway, que dejará de enrutarle tráfico. Un mecanismo de autosanación (self-healing), implementado a través de un script supervisor, se encargará de relanzar la instancia caída.

Para el DB Service, la tolerancia a fallos se basa en la mayoría de quórum de Raft. En un clúster de 5 nodos, el sistema puede tolerar la falla de 2 nodos. Si el nodo líder falla, los nodos restantes iniciarán una nueva elección para designar un nuevo líder y el servicio continuará operando. El sistema es resiliente a fallos parciales, y los nodos que se

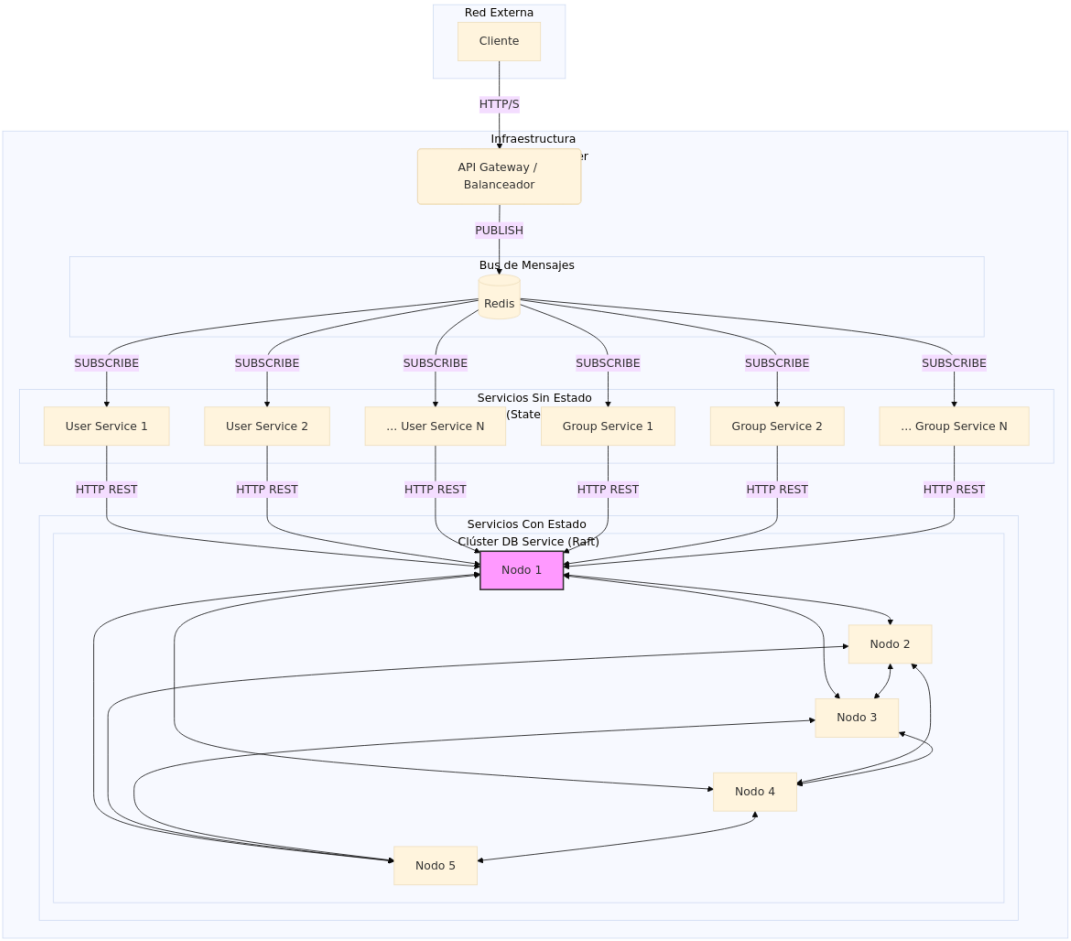

Figure 3: Modelo de Replicación de Servicios y Nodos de Datos

recuperan pueden reintegrarse al clúster sincronizando su estado a partir del log replicado del líder actual.

2.8 Seguridad

La seguridad se aborda en tres frentes. Primero, la comunicación entre el cliente y el API Gateway, así como entre los servicios internos, debe ser cifrada utilizando TLS (HTTPS) para prevenir la interceptación de datos. Segundo, el diseño de microservicios contribuye a la seguridad por aislamiento; un compromiso en un servicio no otorga acceso directo a los datos o la lógica de los demás.

Tercero, la autenticación y autorización se centralizan en el API Gateway. Este valida los tokens de autenticación (por ejemplo, JWT) emitidos a los usuarios tras un inicio de sesión exitoso. Las peticiones a los servicios internos solo son permitidas si provienen del Gateway y contienen un token válido. Los servicios de backend pueden así confiar en que la petición ha sido autenticada, simplificando su lógica y creando un perímetro de seguridad claro.

3 Conclusiones

La arquitectura distribuida propuesta transforma la Agenda Distribuida de un prototipo funcional a un sistema robusto, escalable y resiliente. Mediante la aplicación de principios de sistemas distribuidos, como el consenso Raft para datos stateful, la replicación activa para lógica stateless y el descubrimiento de servicios basado en Gossip, el diseño aborda de manera integral los requisitos de tolerancia a fallos, consistencia y disponibilidad.

La implementación de este diseño no solo cumplirá con los objetivos académicos del proyecto, sino que también proporcionará una base sólida sobre la cual se pueden construir funcionalidades más avanzadas. El trabajo futuro se centrará en la implementación efectiva de los algoritmos de Gossip y Raft, la instrumentación de la seguridad mediante TLS y la realización de pruebas exhaustivas para validar el comportamiento del sistema bajo diversas condiciones de fallo.