Informe Técnico de la Agenda Distribuida

Equipo Agenda_Distribuida

1. Introducción

Este informe documenta la implementación del proyecto *Agenda Distribuida*, un sistema distribuido que gestiona agendas personales y grupales con tolerancia a fallos. El sistema utiliza el algoritmo de consenso Raft para mantener la consistencia de los datos entre múltiples réplicas. A continuación se explican las decisiones de diseño, la arquitectura implementada, los procesos internos, los mecanismos de comunicación, coordinación, nombrado, consistencia, tolerancia a fallas y seguridad, basándose en el código fuente del proyecto y las pruebas realizadas con siete réplicas en Docker Swarm.

2. Arquitectura del Sistema

El desafío principal al diseñar una agenda distribuida es lograr que múltiples servidores trabajen juntos como si fueran uno solo, manteniendo los datos sincronizados y permitiendo que cualquier servidor pueda responder a las peticiones de los clientes.

La solución implementada es una arquitectura homogénea donde todos los contenedores ejecutan exactamente el mismo programa. Esto significa que no hay servidores especializados: cualquier réplica puede convertirse en líder cuando sea necesario, sin cambios en el código. Esta simplicidad facilita el despliegue y la escalabilidad.

El sistema está organizado en varias capas principales:

- **Capa de presentación**: El servidor HTTP/REST (`http_handlers_scaffold.go`) y WebSocket (`websocket.go`) reciben las peticiones de los usuarios y las convierten en operaciones del sistema.
- **Capa de lógica de negocio**: En `services.go` se implementan las reglas de negocio como la creación de grupos, la gestión de agendas y el manejo de notificaciones.
- **Capa de consenso**: Los archivos `consensus.go`, `raft_http.go` y `raft_apply.go` implementan el algoritmo Raft para garantizar que todas las réplicas estén de acuerdo sobre el estado del sistema.
- **Capa de persistencia**: `storage.go` maneja el almacenamiento en SQLite, que actúa como la base de datos local de cada réplica.
- **Capa de descubrimiento y auditoría**: `discovery.go` y `audit.go` permiten que los nodos se encuentren entre sí y registren todas las operaciones importantes.

El despliegue se realiza usando Docker Swarm con siete réplicas distribuidas en al menos dos hosts físicos. Esto garantiza que si un servidor o incluso un host completo

falla, el sistema continúe funcionando. Docker Swarm se encarga de crear las redes virtuales, balancear las peticiones de entrada y supervisar que todos los contenedores estén funcionando correctamente.

3. Organización Distribuida y Roles

Aunque todos los contenedores ejecutan el mismo programa, durante la operación cada uno puede tener un rol diferente dentro del cluster:

- **Usuarios finales**: Interactúan con la interfaz web en `/ui/` y con la API REST. Sus acciones quedan registradas en los logs de auditoría y se autentican mediante tokens JWT.
- **Nodos del cluster**: Son las réplicas del servicio. Cualquiera puede convertirse en líder Raft mediante elecciones, pero solo uno lo es a la vez. Los nodos que no son líderes se llaman seguidores.
- **Nodos manager de Swarm**: Son los hosts que gestionan el clúster Docker. Distribuyen los contenedores entre los diferentes servidores físicos y exponen los servicios al exterior.
- **Operadores**: Pueden consultar los logs de auditoría en `/api/admin/audit/logs` para revisar qué ha ocurrido en el sistema.

3.1. Distribución en Redes Docker

El sistema utiliza una red Docker overlay llamada `agenda_net` para la comunicación entre servidores. Esta red permite que los contenedores en diferentes hosts físicos se comuniquen entre sí como si estuvieran en la misma red local.

Las peticiones de los clientes llegan al cluster a través del modo *ingress* de Docker Swarm, que expone el puerto 8080 de manera balanceada entre todas las réplicas. Internamente, todos los nodos se comunican a través de la red overlay `agenda_net`, lo que permite mantener el tráfico de consenso y descubrimiento separado del tráfico de clientes.

4. Procesos y Patrones de Desempeño

Cada contenedor ejecuta un único proceso Go, pero internamente el programa utiliza múltiples *goroutines* (hilos ligeros de Go) para manejar diferentes tareas de forma concurrente:

- **Servidor HTTP/WebSocket**: Utiliza la librería `gorilla/mux` para manejar múltiples peticiones simultáneamente. Las goroutines permiten atender cientos de conexiones WebSocket sin bloquear el servidor.
- **Motor Raft**: Ejecuta goroutines dedicadas para enviar heartbeats (latidos), iniciar elecciones cuando el líder falla, y replicar entradas del log a los seguidores. Estas goroutines comparten información mediante mutexes para evitar condiciones de carrera.

- **Discovery Manager**: Ejecuta tres bucles en paralelo:
  - Un bucle que consulta DNS tradicional (útil si se necesita integración con sistemas externos).
  - Un bucle que consulta el DNS de Docker para descubrir las réplicas del servicio usando el nombre `tasks.agenda`.
  - Un bucle que intercambia información mediante peticiones HTTP con nodos semilla configurados en la variable `DISCOVERY_SEEDS` (este mecanismo se llama *gossip*).
- **Sistema de auditoría**: Mantiene una cola en memoria para registrar todos los eventos importantes antes de guardarlos en la base de datos.
- **Base de datos SQLite**: Se ejecuta en modo *journaled*, lo que garantiza que las transacciones se completen de forma segura incluso si el programa se detiene inesperadamente.

El patrón de desempeño elegido es cooperativo: en lugar de usar hilos pesados del sistema operativo o procesos separados, se utilizan goroutines que son muy ligeras. Esto reduce el consumo de recursos y permite que el sistema escale mejor al aumentar el número de réplicas.

5. Comunicación

El sistema utiliza diferentes mecanismos de comunicación según quién se esté comunicando con quién:

5.1. Comunicación Cliente-Servidor

Los clientes (navegadores web, aplicaciones móviles, etc.) se comunican con el servidor mediante:

- **REST/JSON**: Todas las operaciones CRUD (crear, leer, actualizar, eliminar) se exponen como endpoints HTTP que aceptan y devuelven JSON. Por ejemplo, `POST /api/appointments` crea una nueva cita.
- **WebSockets**: Para las notificaciones en tiempo real, se utiliza WebSocket. Cuando un usuario acepta una invitación, todos los usuarios conectados reciben la notificación aunque necesitan recargar la página para verla.
- **Autenticación JWT**: Toda petición autenticada debe incluir un token JWT en el header `Authorization: Bearer <token>`. Este token se genera cuando el usuario inicia sesión y contiene información sobre su identidad.

5.2. Comunicación Servidor-Servidor

Los nodos del cluster se comunican entre sí para:

- **Consenso Raft**: Los mensajes de Raft (solicitudes de voto, réplicas de log) se envían mediante HTTP POST a los endpoints `/raft/request-vote` y `/raft/append-entries`.

- **Descubrimiento**: Los nodos intercambian información sobre quién está en el cluster mediante peticiones HTTP a `/cluster/join` y `/cluster/nodes`.
- **Seguridad HMAC**: Todas las peticiones entre nodos deben incluir un header `X-Cluster-Signature` que contiene un hash HMAC-SHA256 del cuerpo de la petición. El secreto compartido está en la variable de entorno `CLUSTER_HMAC_SECRET`. Esto garantiza que solo los nodos autorizados puedan participar en el cluster.

5.3. Comunicación Interna

Dentro de cada nodo, los diferentes componentes se comunican mediante interfaces bien definidas declaradas en `interfaces.go`. Por ejemplo, los servicios de negocio no acceden directamente a la base de datos, sino que utilizan repositorios. Esto permite cambiar la implementación de almacenamiento sin modificar la lógica de negocio.

6. Coordinación

La coordinación entre nodos se logra mediante una implementación del algoritmo de consenso Raft. El objetivo es garantizar que todos los nodos estén de acuerdo sobre qué cambios se han aplicado y en qué orden.

6.1. Flujo de una Operación

Cuando un usuario quiere crear una cita:

1. La petición llega a cualquier nodo del cluster. Si el nodo no es el líder, redirige la petición al líder (código 307).
2. El líder crea una entrada en su log local con un número de índice único y un término (número de ronda de elección actual).
3. El líder envía esta entrada a todos los seguidores mediante `AppendEntries`.
4. Cada seguidor verifica que la entrada anterior coincide con lo que tiene. Si hay diferencias, trunca su log desde ese punto.
5. Cuando la mayoría de los nodos (al menos 4 de 7) confirman que tienen la entrada, el líder la marca como *comprometida*.
6. Solo entonces se aplica la operación a la base de datos SQLite en todos los nodos.

6.2. Mecanismos de Prevención de Divergencias

Para evitar que los nodos tengan datos diferentes:

- Los seguidores comparan el término y el índice antes de aceptar nuevas entradas. Si detectan discrepancias, descartan sus cambios y siguen al líder.
- El middleware HTTP `LeaderWriteMiddleware` garantiza que solo el líder procese operaciones de escritura. Si un seguidor recibe una escritura, la redirige al líder.

- La tabla `raft_applied` registra qué eventos ya se aplicaron, evitando aplicar el mismo cambio dos veces después de un reinicio.
- Todos los eventos importantes se registran en `audit_logs` para poder reconstruir qué ocurrió si hay problemas.

7. Nombrado y Localización

El sistema necesita identificar de manera única a cada réplica y saber cómo contactarla para comunicarse con ella.

7.1. Identificación de Nodos

Cada contenedor recibe un identificador único mediante la variable de entorno `NODE_ID`, que en Docker Swarm se configura como `agenda-{{.Task.Slot}}` (por ejemplo, `agenda-1`, `agenda-2`, etc.). Este identificador se usa en los logs y auditorías para saber qué nodo realizó cada acción.

La dirección de contacto se configura mediante `ADVERTISE_ADDR`, que normalmente es `{{.Task.Name}}:8080`. Esta dirección es la que otros nodos utilizan para comunicarse con esta réplica.

7.2. Descubrimiento de Nodos

El `DiscoveryManager` mantiene actualizada la tabla `cluster_nodes` en la base de datos combinando tres fuentes:

- **Docker DNS**: Consulta periódicamente `tasks.agenda` para obtener las direcciones IP de todas las réplicas del servicio.
- **Nodos semilla**: Si se configura `DISCOVERY_SEEDS`, envía periódicamente peticiones HTTP a esos nodos para anunciar su existencia y recibir información sobre otros nodos (protocolo gossip).
- **DNS tradicional**: Opcionalmente, puede consultar un nombre DNS configurado en `DISCOVERY_DNS_NAME`.

Cada entrada en `cluster_nodes` almacena el identificador del nodo, su dirección, la fuente de descubrimiento y cuándo fue visto por última vez. Si un nodo no se ha visto en más de 2 minutos, se considera desconectado y se retira del `PeerStore`, que es la lista de nodos que Raft utiliza para comunicarse.

7.3. Localización de Datos

Como cada réplica mantiene una copia completa de la base de datos `agenda.db`, localizar un dato es equivalente a contactar cualquier nodo del cluster. El campo `origin_node` en las tablas de citas indica en qué nodo se creó originalmente el dato, lo cual es útil para diagnóstico y posibles optimizaciones futuras.

8. Consistencia y Replicación

El sistema garantiza consistencia fuerte (linearizable) entre todas las réplicas mediante el modelo de log replicado de Raft.

8.1. El Log como Fuente de Verdad

El log de Raft (almacenado en la tabla `raft_log`) es la única fuente de verdad. Hasta que una entrada no esté comprometida por la mayoría de los nodos, no se considera válida. Esto significa que aunque una entrada esté en el log local, no se aplica a la base de datos hasta que la mayoría la confirme.

8.2. Proceso de Replicación

La replicación ocurre en dos etapas:

1. **Replicación del log**: El líder envía cada nueva entrada a los otros seis nodos. Cada entrada contiene metadatos (término, índice, identificador único de evento) y el comando a ejecutar (por ejemplo, çrear cita con estos datos"). El líder espera confirmaciones de al menos cuatro réplicas antes de marcar la entrada como comprometida.
2. **Aplicación a la base de datos**: Una vez comprometida, cada nodo aplica el comando en su base de datos SQLite. De esta forma, las siete copias del dataset permanecen equivalentes.

8.3. Rastreo de Versiones

Para poder diagnosticar problemas y detectar conflictos, las tablas incluyen campos como:

- `version`: Incrementa cada vez que se modifica un registro.
- `origin_node`: Indica en qué nodo se creó originalmente.
- `raft_applied`: Tabla que registra qué eventos ya se materializaron en la base de datos.

8.4. Recuperación tras Reinicios

Cuando un nodo se reinicia, lee su estado desde `raft_meta`, que contiene:

- `currentTerm`: El término actual (número de elecciones que ha visto).
- `commitIndex`: Hasta qué índice del log se ha comprometido.
- `lastApplied`: Hasta qué índice se ha aplicado a la base de datos.

Con esta información, el nodo puede continuar desde donde se quedó, aplicando las entradas faltantes del log.

9. Tolerancia a Fallas

El sistema está diseñado para continuar funcionando incluso cuando algunos componentes fallan.

9.1. Tolerancia en la Capa de Orquestación

Docker Swarm mantiene el servicio con siete réplicas y limita a cuatro el número máximo de réplicas por nodo físico. Esto significa que incluso si un host completo desaparece, el cluster mantiene la mayoría necesaria para operar (al menos cuatro nodos de siete). Swarm automáticamente reprograma los contenedores que fallan en otros hosts disponibles.

9.2. Tolerancia en la Capa de Consenso

Raft garantiza que cualquier subconjunto de cuatro nodos mantenga el quorum (mayoría). Esto significa:

- Si el líder falla, los temporizadores de elección (configurados entre 1,2 y 1,8 segundos de forma aleatoria para evitar elecciones simultáneas) disparan una nueva elección rápidamente.
- El nodo que gana la elección se convierte en el nuevo líder y comienza a enviar heartbeats a los seguidores.
- El middleware de escritura se ajusta automáticamente para redirigir las peticiones al nuevo líder.

9.3. Detección de Nodos Caídos

El `DiscoveryManager` detecta nodos inactivos mediante el campo `last_seen`. Si un nodo no se ha visto en más de 2 minutos, se retira automáticamente del `PeerStore` para evitar intentos de comunicación inútiles que retrasarían las operaciones.

9.4. Reincorporación de Nodos

Cuando un nodo que había fallado se reintegra:

1. Sincroniza la tabla `cluster_nodes` para conocer los demás nodos del cluster.
2. Descarga las entradas faltantes de su log mediante peticiones `AppendEntries` al líder.
3. Aplica las entradas comprometidas que aún no había aplicado a su base de datos.
4. Registra en `raft_applied` qué eventos ya procesó para evitar duplicados.
5. Continúa operando normalmente.

Todos estos eventos quedan registrados en `audit_logs`, permitiendo que los operadores reconstruyan exactamente qué ocurrió durante un failover o la reincorporación de un nodo.

10. Seguridad

El diseño de seguridad abarca tres aspectos principales: la comunicación, la arquitectura interna y la autenticación/autorización.

10.1. Seguridad en la Comunicación

- **Firma HMAC**: Todas las peticiones RPC entre nodos se firman con HMAC-SHA256 usando el secreto compartido `CLUSTER_HMAC_SECRET`. Esto garantiza:
  - Autenticidad: El mensaje proviene de un nodo autorizado.
  - Integridad: El mensaje no ha sido modificado en tránsito.

  Si un nodo no autorizado intenta participar, sus peticiones serán rechazadas por falta de firma válida.
- **TLS opcional**: Si se configuran `TLS_CERT_FILE` y `TLS_KEY_FILE`, el servidor activa TLS (HTTPS) para todas las conexiones. Esto protege contra:
  - Interceptación de tráfico (ataques man-in-the-middle).
  - Observación de datos sensibles en la red.

10.2. Seguridad en el Diseño

- **Separación de responsabilidades**: La capa HTTP no tiene acceso directo a la base de datos. Todas las operaciones pasan por los servicios de negocio y luego por los repositorios. Esto previene inyecciones SQL desde las peticiones HTTP.
- **Registro de auditoría**: Todas las operaciones críticas se registran en `audit_logs`, incluyendo quién realizó la acción, cuándo, y desde qué nodo. Esto permite:
  - Reconstruir qué ocurrió ante un incidente.
  - Detectar acciones no autorizadas.
  - Cumplir con requisitos de cumplimiento normativo.
- **Acceso restringido a auditoría**: El endpoint `/api/admin/audit/logs` requiere además del JWT del usuario, un header adicional `X-Audit-Token` con el valor configurado en `AUDIT_API_TOKEN`. Solo los operadores autorizados pueden consultar estos logs.

10.3. Autenticación y Autorización

- **JWT para usuarios**: Cada usuario obtiene un token JWT al iniciar sesión. Este token contiene su identificador y nombre de usuario, y está firmado para que no pueda ser falsificado. El middleware valida el token en cada petición protegida.
- **Verificación de permisos**: Las acciones sensibles (crear grupos, añadir miembros, etc.) verifican explícitamente que el usuario tenga los permisos adecuados consultando los repositorios. Por ejemplo:

- Solo el creador de un grupo puede eliminarlo.
- Solo usuarios con rango superior pueden modificar miembros de grupos jerárquicos.
- Solo el dueño de una cita puede modificarla o eliminarla.

En conjunto, estos mecanismos reducen significativamente la superficie de ataque del sistema y proporcionan garantías de que las acciones realizadas en el cluster pueden ser rastreadas y auditadas.

11. Conclusiones

La implementación del proyecto Agenda Distribuida demuestra un sistema funcional con consenso distribuido, registro de auditoría completo, descubrimiento automático de nodos y despliegue reproducible en Docker Swarm.

Las pruebas realizadas con múltiples contenedores confirman que la arquitectura es capaz de:

- Tolerar fallas parciales (nodos individuales o incluso hosts completos).
- Mantener la consistencia de los datos mediante el algoritmo Raft.
- Escalar horizontalmente añadiendo más réplicas.
- Proporcionar observabilidad suficiente para diagnosticar problemas y auditar operaciones.

Los posibles trabajos futuros incluyen: completar escenarios de despliegue multi-host, implementar mecanismos de snapshotting para optimizar la recuperación ante fallas prolongadas, y añadir más operaciones al log de Raft (actualmente solo las citas personales pasan por consenso).