Propuesta de Arquitectura - Sistemas Distribuidos
E-Tournaments

Adrián Hernández Santos **411**
Jorge Alejandro Pichardo Cabrera **412**

Enero, 2025

1 Arquitectura

Red Chord base, donde cada servidor realiza función de ejecución, almacenamiento y notificación. Se decide la utilización de Chord, dada su simplicidad y robustez, además de que su diseño de anillo nos permite decidir posteriormente determinadas hipótesis para nuestra replicación y distribución de datos.

1.1 Distribución de servicios

- Servicio de ejecución: Se encarga de ejecutar las partidas asociadas a un torneo, dado el código de un juego y un par de jugadores.
- Servicio de almacenamiento: Almacena los diferentes estados de las partidas en ejecución, así como se encarga de mantener una sincronización de estas con el almacenamiento persistente (instancia de Redis).
- Servicio de notificación: Se mantiene escuchando cambios de estado en las partidas, para notificar de estos a los nodos de la red, y así actualizar los distintos estados.

2 Procesos

Es el problema de cuántos programas o servicios posee el sistema.

2.1 Tipos de procesos

- Proceso de Red (Chord)
- Procesos de partidas (Administrador de ejecución)
- Proceso de notificación

2.2 Organización

El servicio de Chord tiene varias goroutines que se encargan de actualizar la finger table, estar constantemente actualizando los peers y balancear la red. El administrador de ejecución crea una goroutine por cada partida en ejecución, además agrega un channel al proceso de notificación para que este escuche los cambios o eventos que se emitan en el mismo. El proceso de notificación instancia una goroutine principal que comparte un channel con el administrador de ejecución, el cual se encarga de informar los cambios de estado en las partidas que se ejecutan, y permite que el proceso de notificación pueda realizar tanto modificaciones de almacenamiento como broadcasts de estos a los nodos de la red.

2.3 Patrones de diseño

Se utiliza el patrón de goroutines para la creación de hilos/procesos dependiendo de la disponibilidad del SO, gracias a la integración nativa de Go. Además, para la interacción con el almacenamiento persistente se hace uso del patrón async/await para permitir que se liberen recursos del sistema cuando se realizan operaciones I/O y el proceso en cuestión (notificación) pueda realizar tareas de red, por ejemplo, enviar broadcasts de cambios de estado.

3 Comunicación

Se utiliza una combinación de métodos entre los que se encuentran RPC, sockets y patrones de mensajes.

3.1 Tipo de comunicación

- RPC: gRPC
- Sockets: Sockets nativos
- Patrones de Mensajes: ZMQ

3.2 Comunicación cliente - servidor y servidor - servidor

- Cliente - Servidor: Se utiliza gRPC como implementación de RPC para todo lo que es intercambio de datos con el cliente, tales como CRUD de torneos, subida de los códigos de los juegos y jugadores, peticiones de estadísticas, etc.
- Servidor - Servidor:
  - Sockets: El uso de sockets se realiza para todo lo relacionado con la red de Chord, puesto que reduce el tamaño de los paquetes y hace más eficiente y rápido el intercambio de información a bajo nivel, lo que es crucial para el funcionamiento de la red descentralizada.

– Patrones de mensajes: Se utiliza ZeroMQ como gestor de colas de mensajes para intercambiar información a alto nivel entre servidores, como notificaciones, cambios de estados, etc. Principalmente se hace uso del patrón pub-sub, pero también se necesitan utilizar mensajes asíncronos, por ejemplo, para dotar de datos iniciales a un nodo que acaba de entrar a la red o uno que se reincorpora luego de una pérdida de conexión o algún evento crítico similar.

3.3 Comunicación entre procesos

Se utilizan los channels de Go, los cuales crean un buffer de intercambio de datos entre procesos, por los que se pueden enviar objetos de un tipo definido, el cual puede ser un estado o una notificación de algún evento.

4 Coordinación

4.1 Acceso exclusivo a recursos. Condiciones de carrera

Siempre que algún dato crítico, como por ejemplo un estado, se modifica en un hilo, se utiliza un mutex para bloquear su acceso asíncrono y siempre cada actualización va acompañada de un timestamp para evitar que un dato se sobrescriba con alguno de un tiempo anterior.

4.2 Toma de decisiones distribuidas

Se guarda de forma distribuida el nodo que ejecuta una determinada partida, y en caso de que dicho nodo se desconecte en algún momento de la red, se reasigna la ejecución de la partida a otro nodo, replicando dicha decisión a toda la red. Además, se puede intentar un sistema de revocación de decisiones, estableciendo un grupo de mensajes con reglas que deben verificar los nodos antes de aceptar una decisión importante, como por ejemplo esta que se presenta.

5 Nombrado y Localización

5.1 Identificación de los datos y servicios

Nuestro problema tiene los siguientes datos a almacenar distribuidamente:

- Torneos: El estado de un torneo está representado por los participantes (el tipo del jugador y un identificador dentro del torneo), el juego (el tipo de juego) y las partidas que se han efectuado. Su nombre (key) está dado por el $SHA256(UUID)$ donde el $UUID$ es el ID almacenado al crear la instancia del torneo.
- Partidas: Una partida de un torneo tiene la referencia al torneo al que pertenece y su estado, que puede estar dado por el jugador que juega en

este momento, el estado del tablero, una lista de jugadas, etc. Su nombre (key) está dado por $SHA256(IdTorneo : noPartida)$ donde $IdTorneo$ representa el $UUID$ del torneo al que pertenece la partida y $noPartida$ es el ID de la partida dentro del torneo.

5.2 Ubicación de los datos y servicios

Dado que el estado de un torneo potencialmente se actualiza solamente al terminar una partida, se decidió mantener dicha DHT actualizada en todos los servidores, o sea, todos los torneos son replicados en todos los nodos de la red.

Por otra parte, dado el hash de una partida, se puede determinar un nodo que la está ejecutando, lo cual podría guardarse en otra DHT, donde el (key) es el nombre de un objeto partida y el ($value$) el ID del nodo que la está ejecutando. En caso de que por un rebalanceo de la red el nodo se desconecte o cambie de rango, se puede verificar si el estado de la partida no es $FINALIZADA$, entonces se puede cargar el último estado conocido y continuar su ejecución. Aquí, el nodo que ejecuta la partida replicaría el estado a su sucesor y predecesor, garantizando un factor de replicación 3.

5.3 Localización de los datos y servicios

Para localizar los nodos se utilizara un canal UDP multicast, en el cual los servidores se conectan a escribir su ip cada cierto tiempo. Luego tanto los clientes como los servidores nuevos se conectan a escuchar para determinar una direccion con la cual entrar a la red, en el caso de los clientes, la direccion de algun servidor al que hacer peticiones.

5.4 Replicación y Distribución de los datos

Tanto la DHT que contiene los datos de los torneos como la de *ownership* (quién está ejecutando) de las partidas, se replican a los n nodos de la red. Mientras que la DHT con los datos de las partidas está replicada en 3 nodos: el *owner* y sus 2 vecinos, *sucesor* y *predecesor*.

5.5 Confiabilidad de las réplicas de los datos tras una actualización

Dado que el estado de una partida solamente lo actualiza el nodo que la está ejecutando y lo replica a sus vecinos en el anillo, por esa parte la consistencia está asegurada. Existe el caso de que por alguna razón un nodo se desconecte de la red, se reasigne la partida a otro nodo, entonces se debe verificar que en la DHT de *ownership* de una key, sea el nodo que envió la notificación de actualización de estado.

6 Tolerancia a fallas

Es el problema de: ¿para qué pasar tanto trabajo distribuyendo datos y servicios si al fallar una componente del sistema todo se viene abajo?

6.1 Respuesta a errores

Dependiendo del tipo y criticidad de un error se toman determinadas acciones. Por ejemplo, si un nodo falla un $Notify$ a la hora de actualizar la DHT de Chord, se asume que ese nodo no pertenece al anillo y se eliminan todas sus entradas en la tabla.

6.2 Nivel de tolerancia a fallos esperado

Se espera al menos un nivel de tolerancia a fallos de nivel 2, garantizado parcialmente por el diseño en anillo de la red Chord.

6.3 Fallos parciales

En caso de un fallo parcial, gracias a las funciones *stabilize* y $fixFinger$ de Chord, la red luego de un tiempo converge a un estado estable, si no ocurren más fallos antes de que se estabilice nuevamente.

7 Seguridad

7.1 Seguridad con respecto a la comunicación

Se utilizará SSL para el cifrado de las comunicaciones P2P y un cortafuegos sencillo para solamente exponer los puertos del protocolo a los nodos de la subred de servidores, y el puerto del protocolo gRPC a los clientes.

7.2 Autorización y autenticación

Nuestro proyecto no posee autenticación ni autorización, se asume que como los clientes solamente hacen solicitudes de ejecución y no tienen control sobre los datos directamente, no es necesario un sistema de roles o cuentas.