Enrique Ulises Báez Gómez Tagle
0241823@up.edu.mx¿

Métodos de concurrencia y programación multi-hilo/paralelo

1. Métodos usados en el proyecto 🔓

1.1. Go Routines:

- Ejecutamos el servidor gRPC de forma asíncrona dentro de una go routine, para que el programa no se bloquee mientras escucha y procesa las peticiones.
- Entonces ejecutamos las tareas en paralelo mientras el programa sigue corriendo.

1.2. gRPC y Concurrencia gRPC:

- Manejamos las peticiones de los clientes de forma concurrente, pues agregamos una nueva go routine para cada petición, así atendemos varios clientes al mismo tiempo.
- Usamos los famosísimos *flujos de streaming* para el *produce & consume* para poder manejar de forma más fácil la llegada de varios mensajes en *real time*.

1.3. Context Package:

- Usamos el background de context para poder gestionar la producción y consumo de los mensajes, obteniendo un mecanismo para la propagación de valores y cancelaciones entre go routines.
- **Nota:** Esta parte podría mejorarse a algo más pro como timeouts por ejemplo.

2. Posibles Adiciones a Futuro 🚀

2.1. Canales (Channels):

- Podríamos meter canales para que las go routines se comuniquen entre sí. Esto nos ayudaría a sincronizar tareas concurrentes, como el manejo de logs o coordinar mejor la producción y el consumo en paralelo.

2.2. Mutexes:

- Podríamos usar mutexes para evitar problemas cuando varias go routines intenten acceder al mismo recurso al mismo tiempo, como archivos de logs o bases de datos. Así garantizamos que solo una go routine tenga acceso a la vez.

2.3. WaitGroups:

- Sería útil agregar waitgroups para esperar a que varias go routines terminen antes de seguir con otras tareas. Con esto podríamos sincronizar mejor los procesos concurrentes en el programa.

2.4. Context con Cancelación y Timeout:

- Además de `context.Background()`, podríamos implementar un contexto con cancelación y timeout para controlar mejor las tareas concurrentes que necesiten ser canceladas o que puedan tardarse más de lo debido.

2.5. Worker Pools:

- Podríamos implementar un patrón de worker pool para repartir las tareas entre un grupo controlado de go routines. Esto nos permitiría manejar grandes cargas de trabajo de forma eficiente al balancear el procesamiento entre varios «trabajadores».

3. Tecnologías que usan este tipo de patrones 🔧

- **Apache Kafka**:Sistema de mensajería distribuido que utiliza concurrencia y particiones para manejar flujos de datos en tiempo real, permitiendo la producción y consumo simultáneo de mensajes.
- **Redis**: Utiliza modelos de concurrencia para manejar operaciones rápidas en memoria, con soporte para pub/sub y listas que permiten la comunicación eficiente entre diferentes procesos o aplicaciones.
- **MongoDB**: Implementa concurrencia a través de bloqueos de nivel de documento, lo que permite manejar múltiples operaciones de lectura y escritura al mismo tiempo sin que se bloqueen mutuamente.
- **Apache Airflow**: Gestor de flujos de trabajo que ejecuta tareas de manera concurrente en diferentes trabajadores, ideal para la automatización de procesos complejos en paralelo.
- **NGINX**: Servidor web y proxy inverso maneja múltiples conexiones de clientes de forma concurrente utilizando un modelo de eventos asíncrono altamente eficiente.
- **Apache Server**: Utiliza hilos y procesos para manejar múltiples solicitudes concurrentes de los usuarios, lo que permite servir archivos y datos a través de la red sin bloquear el servidor principal.
- **LAMP/WAMP/XAMP**: Estas pilas de software (Apache, MySQL, y PHP) implementan patrones de concurrencia en la base de datos y el servidor web para manejar múltiples conexiones y solicitudes simultáneamente.
- **Elasticsearch**: Motor de búsqueda distribuido que permite realizar consultas de manera concurrente en grandes volúmenes de datos. Utiliza particiones y réplicas para distribuir las solicitudes y mejorar tanto el rendimiento como la tolerancia a fallos.
- **Cassandra**: Base de datos distribuida que maneja grandes volúmenes de datos de forma concurrente, distribuyendo las escrituras y lecturas entre múltiples nodos, garantizando alta disponibilidad y escalabilidad sin un solo punto de fallo.
- **RabbitMQ**: Sistema de mensajería que utiliza colas para manejar múltiples mensajes concurrentes de productores y consumidores. Permite una comunicación eficiente entre aplicaciones distribuidas mediante la implementación de patrones de pub/sub y enrutamiento avanzado.
- **Zookeeper**: Servicio de coordinación distribuido que gestiona la sincronización entre servidores y aplicaciones concurrentes. Utiliza algoritmos de consenso para garantizar que varias instancias de una aplicación accedan a los recursos de manera coordinada.