AlGlobo

Grupo D*
Federico del Mazo - 100029
Javier Di Santo - 101696
Camila Dvorkin - 101109

Facultad de Ingeniería, Universidad de Buenos Aires
[71.59] Técnicas de Programación Concurrente I

26 de octubre de 2021

Informe

Este informe puede ser leido tanto en PDF (gracias a Pandoc) como en HTML (gracias a rustdoc)

Para documentación especifica del código fuente, que excede a este informe, se puede consultar la documentación de la aplicación.

Trabajo Práctico

Este trabajo práctico se forma por dos distintas implementaciones de un sistema de reservas de vuelos a procesar de manera concurrente:

- La primera parte consiste de un servidor HTTP que responde reservas de vuelos, y para cada una levanta distintos hilos.
  - El motor del servidor es actix-web
  - Su código fuente se puede encontrar en `src/threads` (documentación)
  - El servidor se puede levantar con `cargo run --bin threads` y un ejemplo de un pedido de reserva es `curl -i -d '{"origin":"EZE", "destination":"JFK", "airline":"AA", "hotel":true}' -H "Content-Type: application/json" -X POST http://localhost:8080/`
  - Esta implementación tiene pruebas que pueden ser ejecutadas con `cargo test --bin threads` y una prueba de carga para el servidor se puede ejecutar con `./apache-ab-stresstest.sh` que utiliza la herramienta Apache ab
- La segunda parte consiste en leer un archivo CSV con las distintas reservas de vuelo, y para estas ejecutar un sistema de actores que irán procesandolos.
  - El framework de actores utilizados es actix
  - Su código fuente se puede encontrar en `src/actix` (documentación)
  - El programa se puede ejecutar con `cargo run --bin actix`
  - Esta implementación tiene pruebas que pueden ser ejecutadas con `cargo test --bin actix`
- Dentro de `src/common` se encuentran las funciones comunes a ambas implementaciones.

Primera implementación – Hilos

Implementar la aplicación utilizando las herramientas de concurrencia de la biblioteca standard de Rust vistas en clase: Mutex, RwLock, Semáforos (del crate std-semaphore), Channels, Barriers y Condvars.

Estructuras

Flight Reservation En primer lugar, se crea una estructura que representa una reserva de vuelo. A cada vuelo ingresado por consola, se le asignará un ID para ayudarnos a identificarlo. Además, la estructura cuenta con la información necesaria para que el vuelo se pueda reservar con las configuraciones pedidas. Se almacenará su origen y destino, la aerolínea correspondiente a la que se le realizará el requisito y si el pedido incluye o no la reserva de hotel.

```rust
pub struct FlightReservation {
   pub id: i32,
   pub origin: String,
   pub destination: String,
   pub airline: String,
   pub hotel: bool,
}
```

Statistics Estructura que contiene las estadísticas de la aplicación. Por un lado, contamos con un acumulador de tiempo para poder estimar el tiempo promedio que toma una reserva desde que ingresa el pedido hasta que es finalmente aceptada. Por otro lado, un `HashMap` en donde se irán guardando todas las rutas (origen - destino) realizadas para poder llevar una estadística de las rutas más frecuentes.

```rust
pub struct Statistics {
   sum_time: Arc<RwLock<i64>>,
   destinations: Arc<RwLock<HashMap<String, i64>>>,
}
```

Como se puede ver en la estructura, ambas estructuras son `Arc` para que se puedan usar en varios threads. Además, se usa `RwLock` para proveer seguridad a la hora de leer y escribir en las mismas. Esto se debe a que todos los pedidos que ingresan al sistema van a estar intentando acceder a los recursos de estadísticas, es por eso que es necesario el uso de un mecanismo de sincronismo para que no haya conflictos. `RwLock` nos va a permitir tener un escritor (lock exclusivo) o varios lectores a la vez(lock compartido).

AppState Esta última estructura se trata del estado compartido que se compartirá en cada thread que escuche nuevas solicitudes. La estructura contiene: - Las aerolíneas del tipo `Airlines`, que se trata de un mapa de todas las Aerolíneas con webservice disponibles en nuestro sistema. `Airlines` es un `HashMap` de tipo `<String, Arc<Semaphore>>`, en donde la clave es el nombre de la aerolínea. Y el valor es lo que simula ser el webservice, en este caso, un `Semaphore` que nos permitirá controlar la cantidad de solicitudes que se pueden realizar a cada webservice, teniendo en cuenta que cada aerolínea cuenta con un `rate limit`. Este mapa se popular a partir de un archivo `src/configs/airlines.txt`, el cual indica todos los nombres de las aerolíneas junto a los N pedidos que puede responder de forma concurrente. - La estructura de estadísticas `Statistics` para poder acceder y agregar estadísticas a la aplicación. - El `logger sender` para poder enviar mensajes al canal de logs desde cada thread. Para lograr este pasaje de mensajes al canal de logs, se usa un `Sensor` que permite enviar mensajes al otro lado del canal (múltiples consumidores y un solo productor).

```rust
struct AppState {
    airlines: Airlines,
    statistics: Statistics,
    logger_sender: Sender<LoggerMsg>,
}
```

Implementación

Inicialización El main esta compuesto por una serie de threads con diferentes tareas:

- `logger`: El primer thread que se abre es el logger, que se encarga de escribir tanto por consola como en el archivo de logueo, los mensajes que se van a ir recibiendo. Como se explicó previamente, esto está implementado con un `mpsc::channel` con el objetivo que desde cualquier lugar de la aplicación se pueda enviar mensajes con el `Sender` y desde el thread `logger` los mensajes sean leídos por el `Receiver`.
- `http-server`: Se hace uso de Actix web para recibir requests reales por consola, por lo que este thread crea la App con el estado de la aplicación `AppState` y se queda a la espera de requests para resolverlos.
- `keyboard-loop`: Por último, por detrás tenemos al keyboard que se encarga de recibir dos posibles comandos: 'S'/'STATS' que nos permitirá mostrar las estadísticas de la aplicación, y 'Q'/'QUIT' que nos permitirá salir de la aplicación.

Aplicación Una vez que ya tenemos todo el sistema inicializado, nuestro sistema ya está listo para recibir nuevos requisitos. Si todos los parámetros ingresados son correctos, se procede a realizar la reserva. Si no, se muestra un mensaje de error.

La lógica de la aplicación se encuentra en el archivo `src/threads/alglobo.rs` . En primer lugar se abre un nuevo thread para poder ejecutar concurrentemente el request a la aerolínea por un lado y por el otro lado el request al hotel si él mismo lo requiere(en caso de que el pedido incluya el modo de paquete completo).

En el caso de que la reserva sea por paquete, el pedido se envía al webservice del hotel y como sus reservas nunca se rechazan, el tiempo que tarda en procesar la respuesta simplemente se simula con un sleep de un tiempo random.

De forma simultánea, se envía el pedido a la aerolínea correspondiente. El tiempo que demora en realizar el request va a depender de la cantidad de request que soporta la aerolínea concurrentemente, ya que si la misma está respondiendo la cantidad máxima de pedidos, el request de la aerolínea se va a bloquear y deberá esperar a que un pedido anterior termine. Esto está resuelto por el mismo semáforo que solo le va a permitir acceso a los pedidos si su contador interno es positivo, cada pedido que ingresa adquiere el semáforo decrementando en uno el contador, una vez que finaliza el pedido se incrementa el contador desbloqueando un hilo. Además la aerolínea puede aceptar el pedido o rechazarlo (se simula con un random booleano). Si es rechazado, el sistema espera `retry_seconds` segundos para reintentar el pedido. La cantidad de segundos para reintentar es configurable vía variable de entorno `RETRY_SECONDS`. Por último va a depender del tiempo que tarda en procesar el request que también es simulado simplemente con un sleep de tiempo random.

El resultado final de la reserva entonces necesitará que ambos pedidos (hotel y aerolínea) se completen en el caso de ser paquete o únicamente el pedido a la aerolínea. Es decir que no se puede agregar las estadísticas ni finalizar el request hasta que ambos threads hayan finalizado, y eso se resuelve a partir de monitores, esta herramienta nos brinda la posibilidad de esperar hasta que se cumpla una condición, en este caso si se reserva un paquete se debe esperar que ambos pedidos sean completados y sino solamente el request a la aerolínea.

Una vez que se completa el request a la aerolínea, se procede a agregar las estadísticas correspondientes, se suma el tiempo total que tardó en procesar el pedido de principio a fin y se agrega la ruta solicitada(para agregar estas estadísticas, será necesario obtener el lock para poder leer el estado actual de las estadísticas y agregar las nuevas).

Segunda implementación – Actores

Implementar la aplicación basada en el modelo de Actores, utilizando el framework Actix.

Resolución

Actores

StatActor El actor `StatActor` se encarga de manejar las estadísticas de la aplicación. La estructura del actor cuenta con la acumulacion de los tiempos que toman los request, un `HashMap` con las rutas solicitadas y un `HashMap` con los IDs de los request junto con un contador para saber si finalizó su procesamiento.

```
pub struct StatsActor {
  sum_time: i64,
  destinations: HashMap<String, i64>,
  flights: HashMap<i32, i32>,
}
```

Este actor puede recibir un mensaje a la vez del tipo `Stat`. Al recibir este tipo de mensajes, si el request está finalizado(es decir que si se trata de un paquete, finalizó tanto el pedido del hotel como el de la aerolínea), entonces se procede a sumar el tiempo de procesamiento al contador de tiempos totales y se agrega la ruta al `HashMap` de rutas frecuentes. Además imprime por consola las estadísticas hasta el momento que incluyen la cantidad de vuelos, el tiempo total de procesamiento, el tiempo promedio de procesamiento y las 3 rutas más frecuentes.

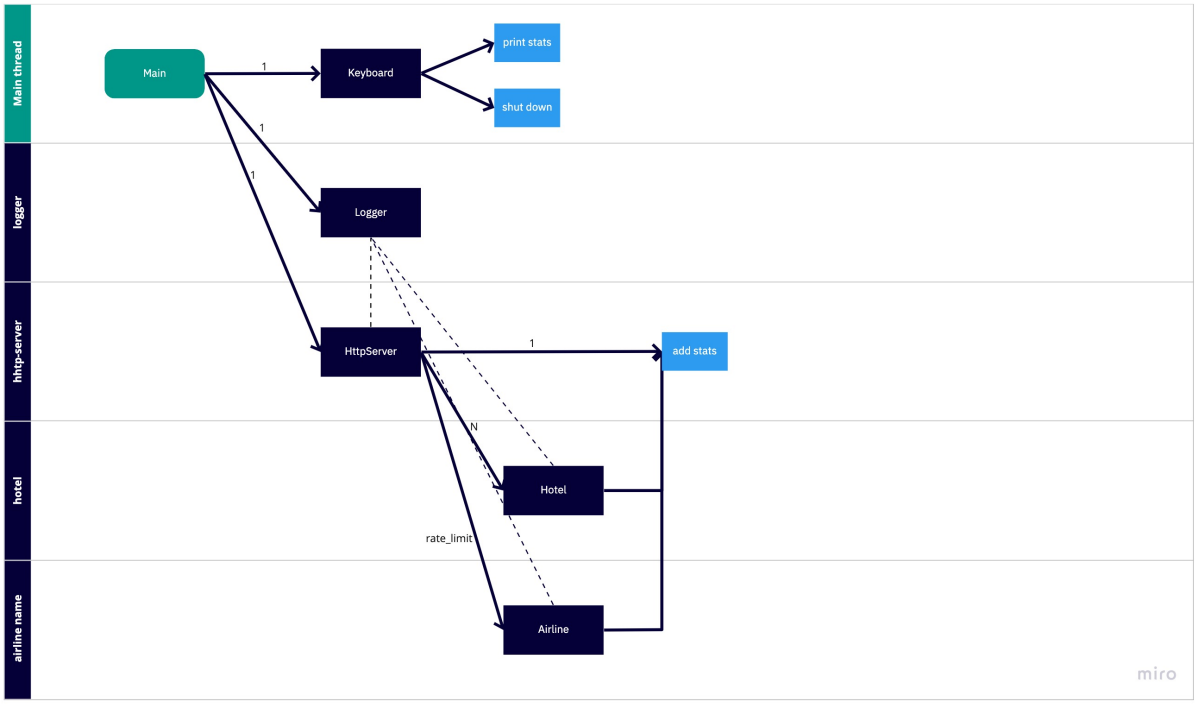

Figure 1: Threads

Por otro lado, puede recibir un mensaje del tipo `FinishMessage` que indica que ya no quedan requests por procesar, por lo que se procede a finalizar la aplicación.

Airline El actor `Airline` simula el webservice de la aerolínea. La estructura únicamente cuenta con la referencia al `StatActor` para poder enviarle los mensajes de estadísticas una vez que termina de procesar el requisito.

A diferencia de `StatActor`, este actor se implementa con un `SyncContext` y esto se debe a que este actor se ejecuta en un `SyncArbitrer` que permite ejecutar `rate_limit` actores simultáneamente. Por lo que, por cada aerolínea, se tiene un `SyncArbitrer` que permite ejecutar N `Airline` simultáneamente acorde a su `rate_limit` establecido en el archivo `src/config/airline.txt`.

Este actor recibe únicamente mensajes del tipo `InfoFlight` y el actor va a simular el procesamiento del request, es decir, va a simular el tiempo que tarda en procesar el request. Este tiempo estará compuesto de la misma manera que esta explicado en la parte A del Trabajo Práctico, es decir que el tiempo va a depender de: cuantos request se pueden procesar simultáneamente, el tiempo que tarda en procesar un request(sleep con duración random) y como puede rechazar los pedidos, se esperarán `retry_seconds` segundos si se rechaza para reintentar el pedido, hasta que se acepte.

Una vez que completa el request, realiza un `try_send` al `StatActor` para enviarle el mensaje de estadísticas correspondiente con el tiempo que tardo en procesar el pedido.

Hotel El actor `Hotel` simula el webservice del hotel. Al igual que la aerolínea, la estructura únicamente cuenta con la referencia al `StatActor` para poder enviarle los mensajes de estadísticas una vez que termina de procesar el request.

El Hotel también es ejecutado en un `SyncArbitrer` que permite ejecutar todos los request en simultáneo.

Este actor recibe mensajes del tipo `InfoFlight` y a los mismos responde simulando el procesamiento del request, es decir, va a simular el tiempo que tarda en procesar el request. Pasado el tiempo de procesamiento (sleep de duración random), se enviará un mensaje al `StatActor` para avisarle que se completó el request y se le mandaran las estadísticas correspondientes con el tiempo que tardo en procesar el pedido.

Mensajes

InfoFlight Mensaje que se envía a los actores `Airline` y `Hotel` para indicar que se recibe un request de vuelo. Está compuesto por la información del vuelo y el tiempo que comenzó a procesarse el request. La respuesta esperada para este tipo de mensajes es vacía.

```
pub struct InfoFlight {
  pub flight_reservation: FlightReservation,
  pub start_time: std::time::Instant,
}
```

Stat Mensaje que se envía al actor `StatsActor` para indicar que finalizó de procesarse el request de vuelo. Está compuesto por el tiempo de procesamiento de un request y `FlightReservation` para conocer la informacion del vuelo. La respuesta esperada para este tipo de mensajes es vacía.

```
pub struct Stat {
  pub elapsed_time: u128,
  pub flight_reservation: FlightReservation,
}
```

Testing

- Para la parte A, se realizan pruebas de volumen gracias a el uso de Actix web, en donde con mayor facilidad se logró enviar muchos pedidos en simultáneo para validar el funcionamiento del programa.
- Se realizan pruebas automatizadas en donde se realizan varias pruebas de una vez, para validar el funcionamiento del programa, implementando nuevamente aquellos métodos que no son determinísticos.

Post Mortem

- try_send()
- condvar por barrieres
- loom
- atixweb en actroes
- exxplicar por qué no usamos stdout para el log (las stats te lo cagan)

Ideas de Todos:

Hablar de correctitud, estado mutable compartido, por que no es fork join, barriers y semáforos

Clavar fotos y docuemntacion de actix web para hablar de los N workers que levanta para escuchar los gets

Una explicación del diseño y de las decisiones tomadas para la implementación de la solución.

Detalle de resolución de la lista de tareas anterior.

Diagrama que refleje los threads, el flujo de comunicación entre ellos y los datos que intercambian.

Clavar un par de screenshots de htop

Diagramas de entidades realizados (structs y demás).