Trabajo Práctico 1
AlGlobo.com: Informe

Técnicas de Programación Concurrente I (75.59)

1^{er} Cuatrimestre 2021
Facultad de Ingeniería
Universidad de Buenos Aires

Integrantes
Mauro Parafati - 102749 - mparafati@fi.uba.ar
Santiago Klein - 102192 - sklein@fi.uba.ar
Tomas Nocetti - 100853 - tnocetti@fi.uba.ar

Repositorio grupal de trabajos prácticos
Página web del grupo

Índice

1. Introducción

Primer trabajo práctico de la materia **Técnicas de Programación Concurrente**, dictada en la Facultad de Ingeniería de la Universidad de Buenos Aires, que consiste en el desarrollo de un sistema concurrente en **Rust** en dos partes: para la primera parte, se debe seguir el modelo de concurrencia de **Estado Mutable Compartido**, mientras que para la segunda parte, se debe seguir el de **Actores**.

Aclaración: los requerimientos son idénticos para ambas partes. Se decide, sin embargo, que la parte A consuma requests desde archivos, mientras que la parte B expone un servidor HTTP para operar sobre las mismas.

Puede accederse al repositorio de **GitHub** donde se desarrolló el del trabajo práctico a través de **este link**.

1.1. Sistema a desarrollar

AlGlobo.com es un nuevo sitio de venta de pasajes online. Gracias al fin de la pandemia ha empezado a crecer fuertemente y necesitan escalar su proceso de reservas.

Para ello desean reemplazar una implementación monolítica actual con un **microservicio en Rust** que se encarge especificamente de este proceso.

1.2. Objetivo

Se deberá implementar una aplicación en **Rust** que modele el sistema de reservas de **AlGlobo.com**. Los pedidos de reserva se leerán desde un archivo (en el caso de la parte A, la parte B utilizará un servidor como ya fue descripto) y el webservice de cada aerolínea se **simulará** con **sleeps aleatorios**, y su resultado utilizando también variables aleatorias (random).

1.3. Requerimientos funcionales

El proceso de reserva se compone de la siguiente manera:

- El pedido de reserva ingresa al sistema e indica aeropuerto de origen, de destino, aerolínea preferida y si la reserva es por paquete o solo vuelo.
- El pedido se debe enviar a la aerolínea correspondiente. Para ello se comunica con un webservice específico de la misma. Por motivos de rate limiting, cada webservice no puede realizar mas de N pedidos concurrentes (es un límite configurable). Tener en cuenta que la cantidad de aerolíneas con las que se comunica el sistema debe poder crecer con un esfuerzo mínimo.
- La aerolínea puede aceptar el pedido o recharzarlo. Si es rechazado, el sistema debe esperar X segundos para reintentar (el tiempo debe ser configurable).
- En el caso de que la reserva sea por paquete, el pedido se debe enviar también al webservice de hoteles. El resultado final de la reserva entonces necesitará que ambos pedidos (hotel y aerolinea) se completen. Por suerte las reservas de hoteles nunca se rechazan.
- El sistema debe mantener estadísticas de las rutas más solicitadas, que luego se utilizan para realizar ofertas. Para ello debe mantener un conteo de la cantidad de reservas realizadas de

cada ruta y poder informar periódicamente las 10 más solicitadas.

- Además el sistema mantendrá estadísticas operacionales. Para ello debe calcular el tiempo medio que toma una reserva desde que ingresa el pedido hasta que es finalmente aceptada.
- Se debe escribir un archivo de log con las operaciones que se realizan y sus resultados.

1.3.1. Modelos de concurrencia

La elección de los modelos de concurrencia a utilizar para la implementación de la solución no es libre; como ya se mencionó, se deben usar ciertos modelos en cada parte:

- **Parte A:** 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.
- **Parte B:** Implementar la aplicación basada en el modelo de Actores, utilizando el framework Actix. Utilizar Tokio HTTP y/o Actix web para recibir requests reales y Apache AB para generarlos en lugar de leerlos de un archivos.

1.4. Requerimientos no funcionales

Los siguientes son los requerimientos no funcionales para la resolución de los ejercicios:

- El proyecto deberá ser desarrollado en lenguaje Rust, usando las herramientas de la biblioteca estándar.
- No se permite utilizar crates externos, salvo los explícitamente mencionados.
- El código fuente debe compilarse en la última versión stable del compilador y no se permite utilizar bloques unsafe.
- El código deberá funcionar en ambiente Unix/Linux.
- El programa deberá ejecutarse en la línea de comandos.
- La compilación no debe arrojar warnings del compilador, ni del linter clippy.
- Las funciones y los tipos de datos (struct) deben estar documentadas siguiendo el estándar de cargo doc.
- El código debe formatearse utilizando cargo fmt.
- Cada tipo de dato implementado debe ser colocado en una unidad de compilación (archivo fuente) independiente.

1.5. Criterios de entrega y de evaluación

Se puede acceder tanto a los criterios de entrega como a los de evaluación en el enunciado original del trabajo disponible en el repositorio.

2. Parte A: Estado Mutable Compartido

El objetivo de esta primera etapa era resolver la problemática utilizando los mecanismos de sincronización de concurrencia vistos durante la primera parte de la cursada.

2.1. Brainstorming

Se comenzó a trabajar pensando en conjunto cual podría ser la mejor solución para encarar el problema, donde surgieron las siguientes ideas.

2.1.1. Hilo por petición y manejador

La idea se basa en **lanzar un hilo por cada petición** que llega al sistema **orquestado por un hilo principal** que se encarga de leer el archivo de peticiones y delegar. De esta manera se optimiza la concurrencia en la resolución de las peticiones pero se agrega una **sobrecarga para la creación de cada uno de los hilos**.

Entendemos que la solución **no es óptima** porque no hay reutilización de los hilos ya creados implicando un consumo innecesario de memoria y CPU. Sin embargo, como ventaja, resulta ser la solución **mas sencilla** de implementar.

2.1.2. Hilo por servicio y canal de llamadas

La idea se basa en **lanzar un hilo manejador de cada aerolínea** y **un hilo para la aerolínea en si**, lo mismo para el hotel. De esta manera, cuando la petición entra, se comunica a través de un *channel* con el manejador correspondiente y este con el servicio. La comunicación en general se maneja toda con *channels*.

Esta solución **se asemeja al modelo de actores** y resultaba bastante **compleja** de implementar. Si bien se trata de una solución mas *performante* por la reutilización de los hilos, también trae consigo el inconveniente de que el único hilo manejador por servicio podría (eventualmente) resultar un cuello de botella.

2.2. Solución implementada

Finalmente se decidió ir por la solución **"Hilo por petición y manejador"**, buscando de esta manera probar varios de los mecanismos vistos en clase y no solo uno, y aprovechando que en la segunda parte se implementaría la otra solución.

Como se mencionó previamente, cada petición procesada tendrá asignada su correspondiente **hilo de ejecución**. Desde allí consumirá los servicios correspondientes *(aerolínea/hotel)* hasta obtener un resultado.

Para los *web services* (servicios correspondientes a las aerolíneas y al hotel) se utilizó una estructura del tipo `semáforo` para controlar el *rate-limit* entre los distintos hilos que consumían la misma aerolínea. Para el caso de los logs y las métricas se uso un `channel` como mecanismo de sincronización que permitía tener estos servicios corriendo en hilos separados.

2.2.1. Entidades

Con la ya detallada solución en mente, se plantearon las siguientes entidades:

- `Dispatcher`: su responsabilidad radica en el procesamiento de las peticiones (lectura del archivo) y la delegación lógica en el `RequestHandler`.
- `RequestHandler`: actúa como intermediario entre el `Dispatcher` y el/(los) `WebService`*(s)*. Se encarga de lanzar el hilo donde se procesara la petición y brindar los recursos necesarios para su correcta ejecución. Como siempre se hace una reserva aérea, el hilo que lanza por cada request **simula el fetch** a un *web service* real de una aerolínea. En el caso de que el pedido sea por un paquete, este hilo, a su vez, *spawneará* otro que simulará el fetch al web service del hotel. Asimismo se encarga de comunicarse con los servicios de `Logs` y `Métricas`. Finalmente es el encargado de esperar la resolución de la petición y hacer el *join* con el hilo principal.
- `WebService`: su responsabilidad radica en simular la petición al servicio de aerolínea/hotel. Es el encargado de **mantener el rate-limit** para cada uno de los servicios, utilizando el semáforo.
- `MetricsCollector`: servicio corriendo en simultaneo que se encarga de la recolección de métricas. Esta modelado con dos hilos que sincronizan mediante una estructura del tipo `RwLock`, uno para recolección y el otro para el output periódico por pantalla. Sincroniza con el `RequestHandler` utilizando una estructura del tipo `channel`.
- `Logger`: servicio corriendo en simultaneo que se encarga del **output a un archivo y por pantalla de los logs** del sistema. Sincroniza con el `RequestHandler` utilizando una estructura del tipo `channel`.

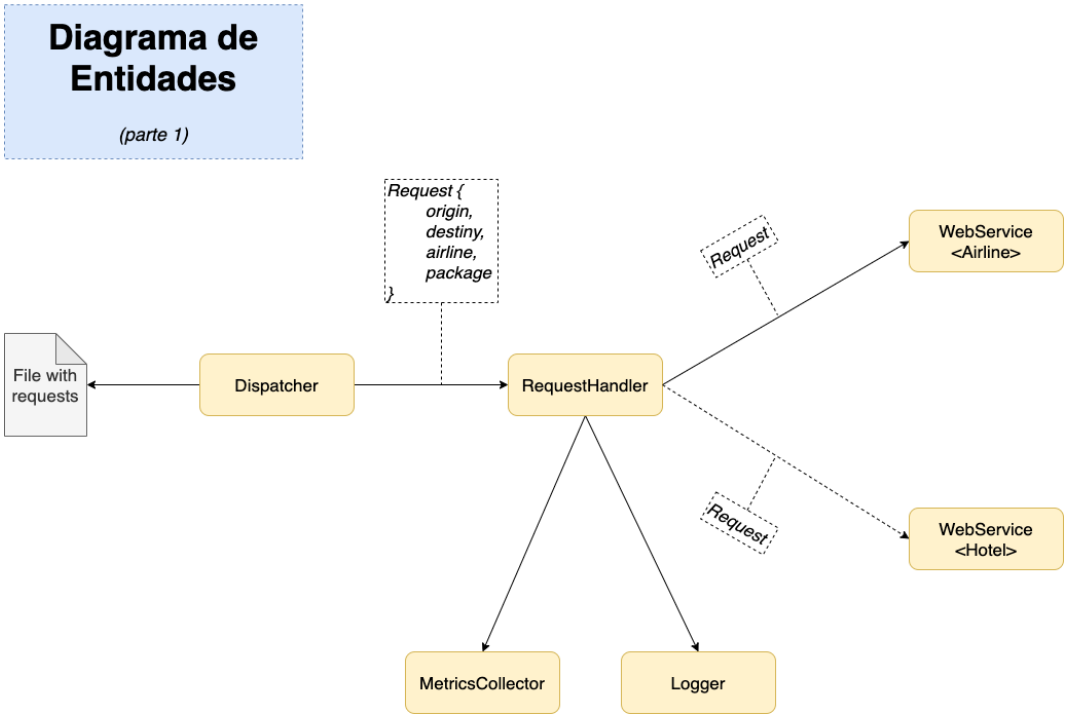

Figura 1: Diagrama de entidades. Entidades identificadas en el sistema .

2.2.1.1 Comunicación

Se muestra en el siguiente diagrama como se plantea el ciclo de vida y la comunicación de las entidades durante la ejecución del programa.

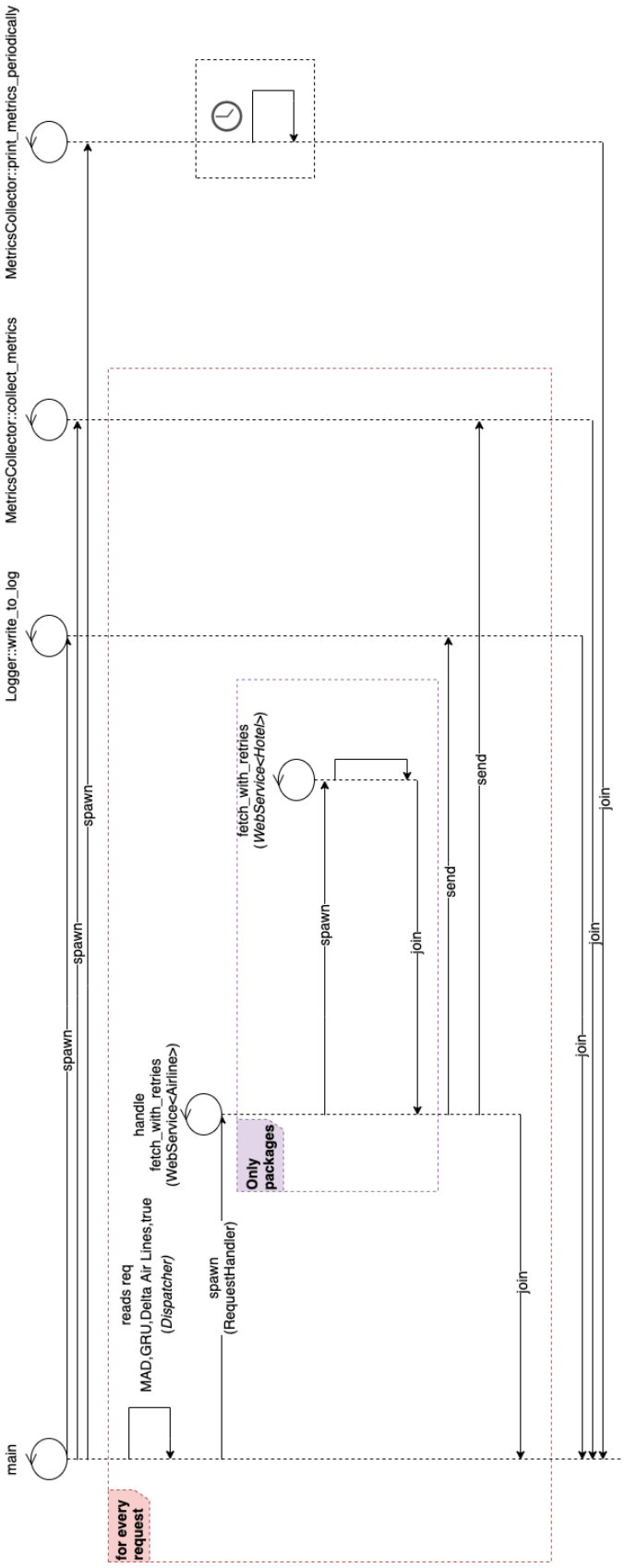

Figura 2: Ciclo de vida y comunicación de entidades

3. Parte B: Actores

Para la parte B la consigna especificaba claramente la necesidad de modelar la problemática con **Actores**, para lo cual se utilizó el framework *Actix*.

En este caso, se implementó un servicio que expone una **REST API**, utilizando *actix-web*. Entonces, la fuente de requests son las llamadas que recibe dicho micro servicio, en vez de la lectura de un archivo, como en la primera parte.

3.1. Solución implementada

Para la resolución de la segunda parte, en una primer instancia de trabajo se definieron los **actores** a diseñar, así como los mensajes con los que se comunicarían entre sí a fin de resolver el problema.

3.1.1. Actores

A continuación, se realiza una breve descripción de los actores utilizados:

- `RequestHandler`: es el encargado de recibir las requests por parte de la API (`POST /request`) y despacharlas al `WebServiceDispatcher` correspondiente. Asimismo, notifica al `StatusService` del ingreso de una nueva request.
- `WebServiceDispatcher`: existe uno por cada `WebService` (de aerolíneas y de hoteles). Su función es implementar el rate limit y efectuar, cuando sea posible, el fetch de las reservas al `WebService`. Para ello, almacena en su estado una cola de requests pendientes, así como el número de requests que se pueden hacer en cada momento.
- `WebService`: simula un servicio de procesamiento de reservas aéreas u hoteleras. Recibe mensajes por parte de su `WebServiceDispatcher`.
- `StatusService`: actor encargado de almacenar el estado de las requests que llegaron al sistema. Los usuarios podrán utilizar la API REST para consultar el estado de las mismas, mediante `GET /request/:id`, utilizando el `id` provisto como respuesta al *POST* correspondiente. Almacena un `HashMap` con el estado de cada request, indexado por su `id`.
- `MetricsCollector`: actor encargado de almacenar las métricas generales de procesamiento de requests. Almacena, entre otras variables, un `HashMap` con el conteo de rutas, indexado por el nombre de las mismas.
- `Logger`: imprime por pantalla y/o almacena mensajes en un archivo de log.

3.1.2. Mensajes

Se realiza una breve descripción de los mensajes que acepta cada actor:

- `RequestHandler`:
  - `HandleRequest`: mensaje enviado por el controller de (POST /request) para iniciar el procesamiento de una request.
- `WebServiceDispatcher`:

  - `HandleBook`: mensaje enviado o bien por el `RequestHandler` o bien por el mismo `WebServiceDispatcher`. Efectúa el fetch de la reserva al `WebService` en caso de que se pueda, o bien la encola en caso de que el rate limit se haya alcanzado.
  - `FetchSucceeded`: mensaje enviado por el `WebService` para notificar que una request fue procesada satisfactoriamente, y así poder o bien liberar un slot del rate limit si no hay request pendientes, o bien realizar la próxima request encolada. Además, notifica al `StatusService` de la completitud de la misma.
  - `FetchFailed`: mensaje enviado por el `WebService` para notificar que el procesamiento de una request falló. Ante este evento, el handler se duerme el tiempo que corresponda (sin bloquear el Event Loop, obviamente), y luego el actor se manda a sí mismo nuevamente el mensaje `HandleBook` con la request fallida.

- `WebService`:
  - `Book`: mensaje enviado por el `WebServiceDispatcher` para el procesamiento de una reserva. Al recibirlo, este actor simula el procesamiento mediante un sleep (no bloqueante del Event Loop) y una respuesta de éxito o falla según una probabilidad determinada, ante la cual envía el mensaje correspondiente al `WebServiceDispatcher` según el resultado.

- `StatusService`:
  - `NewRequest`: mensaje enviado por el `RequestHandler` para registrar la nueva request, con el estado pendiente.
  - `BookSucceeded`: mensaje enviado por un `WebService` (de aerolínea u hotel), para notificar el procesamiento correcto del pedido.
  - `GetStatus`: mensaje enviado por el controlador del endpoint `GET /request/:id` en pos de obtener el estado de una request.

- `MetricsCollector`:
  - `LogMetrics`: mensaje que el actor se envía a sí mismo períodicamente, luego de un sleep (que no bloquea el Event Loop), en aras de loggear en las métricas.
  - `MetricsMessage`: mensaje enviado por el `StatusService` para notificar de la completitud de una request, y el tiempo demorado de la misma.
  - `GetMetrics`: mensaje enviado por el controlador del endpoint `GET /metircs` en pos de obtener las métricas del sistema.

- `Logger`:
  - `LogMessage`: mensaje enviado por casi la totalidad de los actores para imprimir en pantalla y/o almacenar en el archivo de logs un mensaje.

3.1.3. Diagrama del sistema

Se incluye a continuación un diagrama representativo del sistema desarrollado:

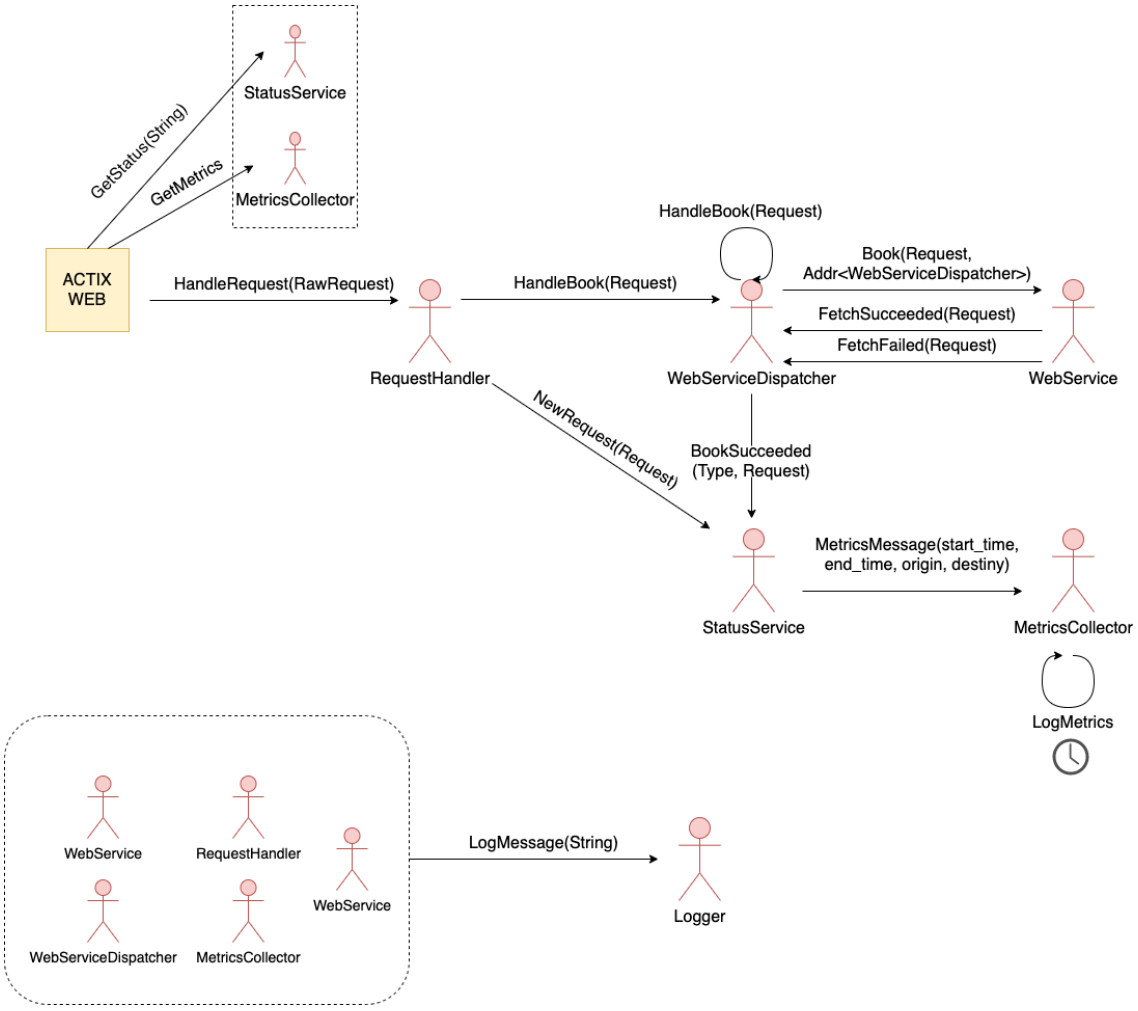

Figura 3: Actores y Mensajes

3.2. Puntos de mejora

Un detalle a considerar es que el sistema actualmente hace un manejo demasiado *naive* en el caso en que cuando un actor intenta enviar un mensaje a otro, este no puede encolarse en la cola de mensajes del destinatario (sea por el motivo que sea: cola de mensajes del otro actor llena, por ejemplo). Actualmente, se tomó el siguiente *approach*:

- Para la comunicación entre los actores del sistema, en general se decidió hacer un **panic** para cortar la ejecución, ya que no debería pasar nunca que se acumulen mensajes sin procesar en una casilla.
- Para las requests recibidas, si sucede un error al intentar hacer el *dispatch* de la misma, en este caso se retornará el error al usuario sin alterar el estado del cliente. Esto contempla el caso esperado de saturación del servidor.

Más allá del enfoque que se adaptó, se entiende que en un ambiente productivo se deberían contemplar estas situaciones y **reaccionar** a ellas aplicando distintas estrategias según el caso.

Por ejemplo, si ya se envió el mensaje de request al hotel pero falla a la hora de enviarlo a la aerolínea, la request fallaría. Sin embargo, la reserva al hotel estaría confirmada, siendo un comportamiento no esperado. Una posible solución podría ser tomar un enfoque **transaccional** que permita hacer el ***rollback*** de la transacción (deshacer ciertas acciones) en caso de que la misma no termine satisfactoriamente, para de esta forma no dar lugar a inconsistencias en el sistema.

4. Conclusiones

Gracias al desarrollo del presente trabajo práctico, se posibilió la puesta en práctica de las distintas herramientas aprendidas a lo largo del curso, y se adquirió una gran cantidad de práctica tanto como con los distintos modelos de concurrencia, así como con el lenguaje de programación Rust (y su *particular* entorno de desarrollo).

A su vez, se considera muy valiosa la experiencia ganada en **lenguajes fuertemente tipados**. Como consecuencia, se observó que la velocidad de desarrollo en el lenguaje disminuyó notoriamente de forma muy pronunciada desde un primer valor tal vez elevado, llegando a desarrollar features nuevas en muy poco tiempo.

Por último, se destaca también el escenario planteado por la situación problemática, que permitió tener numerosas discusiones muy formativas sobre distintos posibles problemas futuros (incluso aunque no sean contemplados en este trabajo), y como se podrían solucionar. El **desarrollo en equipo** fue un punto muy favorable que permitió **potenciar el aprendizaje**.