FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# Trabajo Práctico 2
# AlGlobo: Sistema de Pagos
# 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

Segundo 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 compuesto por distintas entidades independientes que se comunican entre sí mediante sockets para llevar a cabo el procesamiento de una serie de requests.

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

## 1.1. Sistema a desarrollar

Continuamos desarrollando el backend de AlGlobo.com. En este caso vamos a trabajar con el sistema de procesamiento de pagos.

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

## 1.2. Objetivo

Se deberá implementar un conjunto de aplicaciones en Rust que modele el sistema de procesamiento de pagos de AlGlobo.com.

Se debe implementar un proceso para cada una de las entidades intervinientes y estas se comunicarán entre sí por sockets.

Se debe poder simular la salida de servicio de cualquiera de los procesos y réplicas de forma aleatoria o voluntaria, mostrando que el sistema en su conjunto sigue funcionando.

## 1.3. Requerimientos funcionales

El proceso de pagos se compone de la siguiente manera:

- Intervienen 4 entidades: AlGlobo.com, el banco, la aerolínea y el hotel.
- Existe una cola de pagos a procesar que se lee desde un archivo.
- AlGlobo.com debe coordinar el pago informando el monto a cobrar a cada entidad de forma concurrente.
- Cada entidad puede aleatoriamente procesar correctamente el cobro o no.
- Si alguna falla, se debe mantener la transaccionalidad y por lo tanto revertir o cancelar apropiadamente.
- Las fallas se guardan en un archivo de fallas para su posterior procesamiento manual. Debe implementarse una utilidad que permita reintentar manualmente cada pedido fallado.
- El sistema de AlGlobo.com es de misión crítica y por lo tanto debe mantener varias réplicas en línea listas para continuar el proceso, aunque solo una de ellas se encuentra activa al mismo tiempo. Para ello utiliza un algoritmo de elección de líder y mantiene sincronizado entre las réplicas la información del la transacción actual.

## 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. Modelado del sistema

Para la resolución del problema planteado, se implementó:

- un servicio escalable para la entidad *AlGlobo* (y sus réplicas), dividido en un *Control Plane* y un *Data Plane*
- un servicio genérico para las entidades *Aerolínea*, *Banco*, y *Hotel*.

Se utilizaron *docker* y *docker-compose* para la construcción y ejecución de los servicios, con una network interna y externa, y un volúmen para compartir los archivos. Asimismo, se diseñaron scripts para automatizar la ejecución de los comandos más frecuentes.

Los servicios se comunican entre sí mediante *Sockets UDP*, en el que se intercambian información en base a dos protocolos:

- un primer protocolo correspondiente al intercambio de mensajes del *Control Plane*, que permite a la detección de nodos *AlGlobo* en ejecución y llevar a cabo el algoritmo de selección de líder
- un segundo protocolo para implementar el algoritmo **Two Phase Commit**, y comunicar las transacciones y las respectivas acciones entre el líder *AlGlobo* por un lado, y la *Aerolínea*, el *Banco*, y el *Hotel* por el otro.

A continuación, analizaremos más en profundidad cómo es que fueron implementados los diferentes servicios.

El sistema modelado funciona bajo las hipótesis de que los servicios siempre funcionan (o se caen por intervalos finitos únicamente, pero no perdiendo el estado almacenado en memoria), y de que los datagramas UDP no son fraccionados (y llegan en orden).

# 3. AlGlobo

La entidad de AlGlobo es nuestra entidad principal. Se modeló con el objetivo de obtener lo más similar a un Nodo independiente que podría ser corrido en cualquier máquina, siempre y cuando esté conectado a la red interna del resto de nodos.

La lógica de un Nodo se encuentra dividida en claras capas de abstracción que favorecen no sólo a distintos atributos de calidad del sistema (como la modificabilidad por ejemplo), sino que también generan que el proceso de desarrollo sea paralelizable.

## 3.1. Orquestador: Fail Fast

La primer capa de abstracción es un **Orquestrador Naive** que se encarga de reiniciar el sistema desde cero cada vez que el mismo falla.

Esto nos permitió tomar un approach de ***Fail Fast*** dentro de cada nodo, es decir, ante un error fallar sin necesidad de manejar el error (salvo casos excepcionales).

El diseño elegido es posible ya que justamente la capa superior se encargará de reiniciar nuestro estado, y una réplica podrá tomar el trabajo de líder mientras tanto.

## 3.2. Control Plane

La segunda capa de abstracción, **Control Plane**, es la capa que se encarga de coordinar a los distintos nodos disponibles en la red para llevar a cabo el proceso eligiendo el líder cuando es necesario.

En esta capa se implementa el **Algoritmo de Selección de líder Bully**, mediante el cual cada vez que el líder no responda, se comenzará una elección para que una replica tome el control del flujo principal. Una salvedad a mencionar es que se decidió que la elección la gane el Nodo con el menor ID, ya que dada la naturaleza auto-incremental de los ids asignados por el directorio, esto permite evitar que si un nodo que presenta reiteradas fallas en su máquina sea siempre elegido líder (puesto que cada vez que resetea, tendrá un ID más alto, y perderá cualquier elección).

En la Control Plane se decidió utilizar **Sockets UDP** con timeouts, puesto que no hay problema si se pierde un paquete, el propio algoritmo se encargará de, en régimen, normalizar la situación llevando a cabo las elecciones que sean necesarias.

## 3.3. Data Plane

Por último, corriendo en el más bajo nivel tenemos la capa de **Data Plane**.

### 3.3.1. Funcionalidad

Esta capa implementa la lógica de **lectura** y **procesamiento de transacciones**, **envío de mensajes** a los **servicios** correspondientes y **guardando en el log** los resultados respectivos.

Consta de dos hilos:

- el hilo principal, encargado de leer las transacciones del archivo de transacciones a realizar, procesar las mismas, haciendo los llamados correspondientes a los servicios (con las acciones *Prepare*, *Commit*, y *Abort*, según el caso.
- el hilo secundario, encargado de recibir las respuestas de los servicios.

Ambos hilos comparten referencias a un *HashMap* de respuestas, que almacena las respuestas de los servicios para la transacción que se está procesando. El acceso a dicho *HashMap* está protegido por un *Mutex*. Además, se hace uso de una **condition variable**. El hilo principal, luego de broadcastear un mensaje determinado a todos los servicios, se queda esperando en esa *condition variable* a que se reciba la respuesta de todos los servicios (o bien a un timeout porque no se recibe la respuesta de algún servicio). El hilo secundario, por su parte, almacena en ese *HashMap* de respuestas las respuestas (valga la redundancia) de los servicios a medida que las va recibiendo.

También comparten un mismo *Socket UDP*, con la salvedad que el hilo principal hace uso del canal de envío y el hilo secundario hace uso del canal de recepción, que son thread safe.

Adicionalmente, se utiliza otra variable, protegida por un *Mutex*, con el id de la transacción actual que se está procesando. De esta manera, el hilo secundario puede descartar las respuestas de los servicios que no corresponden a la transacción actual (que es la única de interés).

Para ello, en primer lugar, hicimos una distinción dentro del mismo *DataPlane*.

- Por un lado, tenemos un servicio automatizado que se levanta junto a los otros servicios en un mismo *docker-compose* (con el script *run.sh*, que es el que se escala junto a las réplicas y participa propiamente en el algoritmo de selección de líder. Este servicio loguea las transacciones fallidas en un archivo de log (compartido entre las réplicas) y, procesa las transacciones del archivo *payments.csv*, y, a medida que va procesando las transacciones, las elimina del archivo de transacciones a procesar.
- Por otra parte, tenemos un servicio manual, que se levanta aparte (con el script *run_ manual.sh*), pero que se conecta a la red expuesta por el docker-compose anterior. Este servicio se utiliza para reintentar las transacciones fallidas, permitiendo ingresar al usuario por línea de comandos las transacciones a reintentar, en formato csv. No persiste información alguna, sino que muestra el output por consola, por lo que en ese sentido es ortogonal a las réplicas de antes, e independiente.

No obstante de ello, ambos servicios (automatizado y manual) comparten la misma lógica de código, con la única excepción de que el primero lee las transacciones a realizar del archivo *payments_ csv* y persiste el resultado de la transacción, mientras que el segundo lee las transacciones a realizar de *stdin* y no persiste el resultado de la transacción.

### 3.3.2. Estados de una transacción

Los estados posibles en los que puede estar una transacción son:

1. **Prepare**: estado inicial.
2. **Commit**: transacción procesada satisfactoriamente.
3. **Abort**: transacción abortada.

Una transacción no termina de ser procesada en tanto y en cuanto *AlGlobo* no reciba un mensaje de *Commit* o *Abort* por parte de **todos** los servicios.

### 3.3.3. Ciclo de vida de una transacción

El *ciclo de vida de una transacción* es el siguiente:

1. Se lee la transacción del archivo *payments.csv*
2. Se escribe en el log *Prepare* y se envía el mensaje *Prepare* a los servicios, y aquí pueden suceder tres cosas:
   *a*) **Todos** los servicios **responden** con el mensaje **Prepare**, por lo que se procede a committear la transacción.
   *b*) **Todos** los servicios **responden**, pero algún servicio responde con el mensaje **Abort**, por lo que se procede a abortar la transacción.
   *c*) **Algún** servicio **no responde**. En este caso, se reintenta hasta 2 veces más enviar el mensaje *Prepare*, hasta que se de alguna de las dos condiciones anteriores. En caso de seguir existiendo un servicio que no responde, se procede a abortar la transacción.
3. Del paso anterior surgen dos resultados posibles: **Commit** o **Abort**. En ambos casos, se escribe en el log el resultado, y se ingresa en un loop que envía el resultado a los servicios hasta obtener respuesta de todos ellos. Consta resaltar que podemos hacer esto bajo el supuesto de que los servicios están disponible permanentemente

   *Nota*: si bien nuestro sistema tolera fallas momentáneas en los servicios, es necesario que en algún momento vuelvan a estar funcionales, puesto que sino tendríamos un loop infinito en este paso).
4. Fin del ciclo de vida de una request. Se continúa procesando la próxima request del archivo.

### 3.3.4. Modelado de estado compartido entre réplicas

Como fue previamente mencionado, para modelar el estado compartido entre réplicas del servicio *AlGlobo*, se utilizó un archivo de log, en el cual el coordinador escribe el estado de la transacción actual. Este archivo es conocido por todas las réplicas, pero solamente es leído y escrito por el coordinador activo. Contiene únicamente el estado de la última transacción, y siempre se escribe en el log antes de realizar acción alguna (técnica **Write Ahead Log**).

Como las transacciones son borradas del archivo *payments_ csv* a medida que finaliza su ciclo de vida, una réplica, si debe tomar el rol de líder, sabe cuál es la próxima transacción a procesar porque es la primera de este archivo (ya que las ya procesadas fueron eliminadas). Asimismo, para conocer el estado de la última transacción, lee el archivo de log compartido, y en base a esa acción continúa el ciclo de vida del request.

### 3.3.5. Fallas

Para probar el funcionamiento del algoritmo de selección de líder, se introdujo en el servicio *AlGlobo* fallas aleatorias, que provocan la destrucción de las réplicas (de manera independiente), y, en particular, del nodo líder.

Además, se puede optar por hacer un *stop* de los containers, y la reacción del sistema será similar.

# 4. Directorio

Se trata de una entidad especial que se diseñó para actuar a modo de "DNS" entre nuestros nodos. Cada vez que un nuevo nodo quiere unirse a la red de réplicas, debe comenzar solicitando el registro al directorio, quien se va a encargar de asignarle un ID de forma dinámica para que utilice por el resto de su ejecución.

A su vez, el directorio se compromete a actualizar a cada nodo cada vez que una réplica se une o se da de baja del sistema. Para esto, se establece inicialmente una conexión TCP que se mantendrá activa durante toda la ejecución. A través de este canal TCP, el directorio **broadcastea** las nuevas conexiones al Nodo, permitiendole al Nodo tomar un enfoque **no bloqueante** al respecto: utilizando un socket no bloqueante, puede consultar si hay nuevos mensajes de broadcast en su buffer y en caso negativo, seguir con su ejecución, evitando tener que levantar otro hilo de ejecución para comunicarse con el Directorio.

Un detalle no menor es que se prestó especial cuidado en la implementación a que el directorio se encuentre lo más actualizado posible, por lo que cada vez que un proceso se registra, se detectan los procesos que cerraron su conexión TCP para darlos por muertos y poder reutilizar esa dirección o el propio ID, y avisarle a los nodos activos. Un gran beneficio de esta decisión es que se ahorran muchos mensajes de red en el algoritmo Bully.

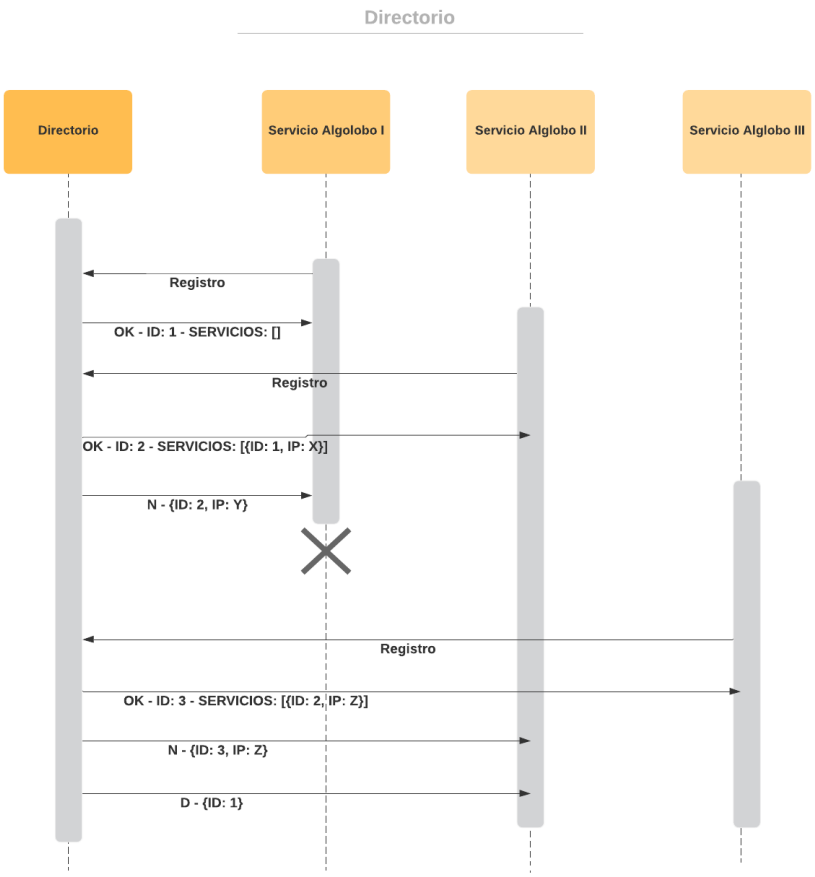


Figura 1: Diagrama de secuencia del comportamiento del directorio con las replicas de Alglobo .

# 5. Servicios

Para implementar los servicios **Aerolínea**, **Banco**, y **Hotel**, se creó un servicio genérico.

Este servicio genérico cuenta con un *HashMap* en memoria que funciona como un log de transacciones. Así, almacena <tx_id, Action>, siendo Action la última respuesta envíada al servidor (*Prepare*, *Commit* o *Abort*).

Con la ayuda de este mismo log es que los servicios son **idempotentes**, lo cual es indispensable para, ante una misma transacción, no realizar reserva/liberación de recursos de más. Por ello, ante un mensaje nuevo, el servicio primero se fija si tiene loggeada la transacción. Si la acción del mensaje coincide con la del log (o, excepcionalmente, recibe un *Prepare* pero tiene un *Commit*), no se realiza procesamiento alguno y se envía el mensaje con el estado correspondiente.

En cambio, si la acción no coincide con la del log, se efectúa el procesamiento correspondiente. Este "procesamiento"depende de la acción pedida:

- **Prepare**: si la acción recibida es *Prepare*, aleatoriamente el servicio responderá *Prepare* o *Abort*. En caso de responder *Prepare*, reservará los recursos necesarios para la transacción, y en el caso de *Abort* no hará nada (se aborta directamente).
- **Commit**: si la acción recibida es *Commit*, se efectuará el consumo definitivo de los recursos reservados, y se responderá con *Commit* nuevamente.
- **Abort**: si la acción recibida es *Abort*, se liberarán los recursos si es que fueron reservados previamente en el *Prepare*.

Este mecanismo se basa en la premisa de que estos servicios no caerán nunca, puesto que sino se perdería el estado que se almacena en el log en memoria.

A continuación se muestran algunos casos de secuencia entre la instancia líder de Alglobo y los servicios Banco, Hotel y Aerolinea.

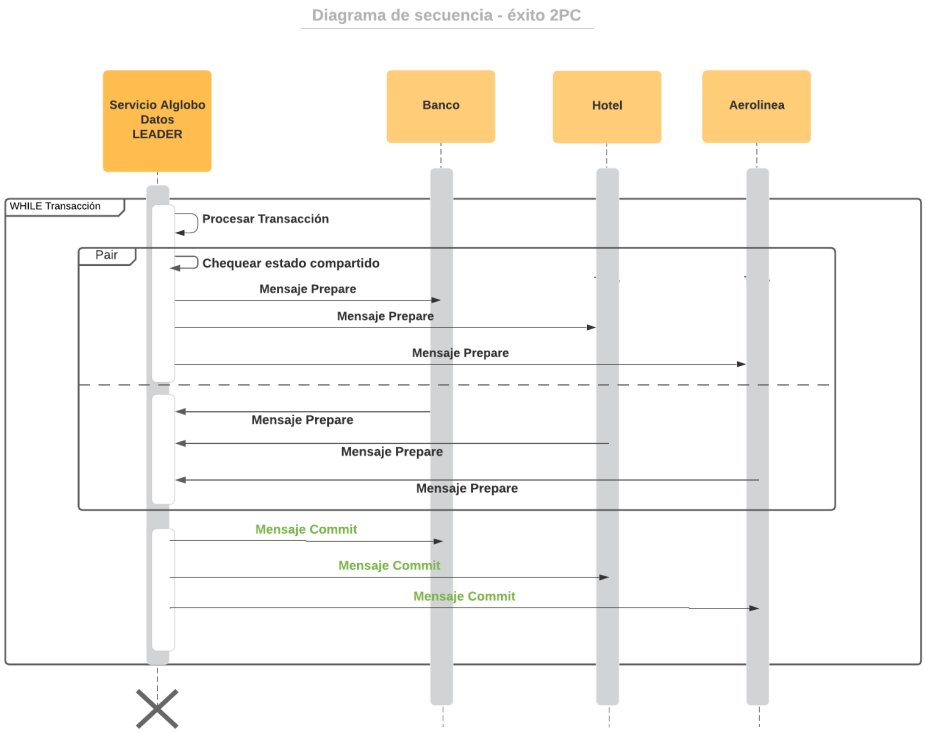


Figura 2: Diagrama de secuencia Two Phase Commit - Success .

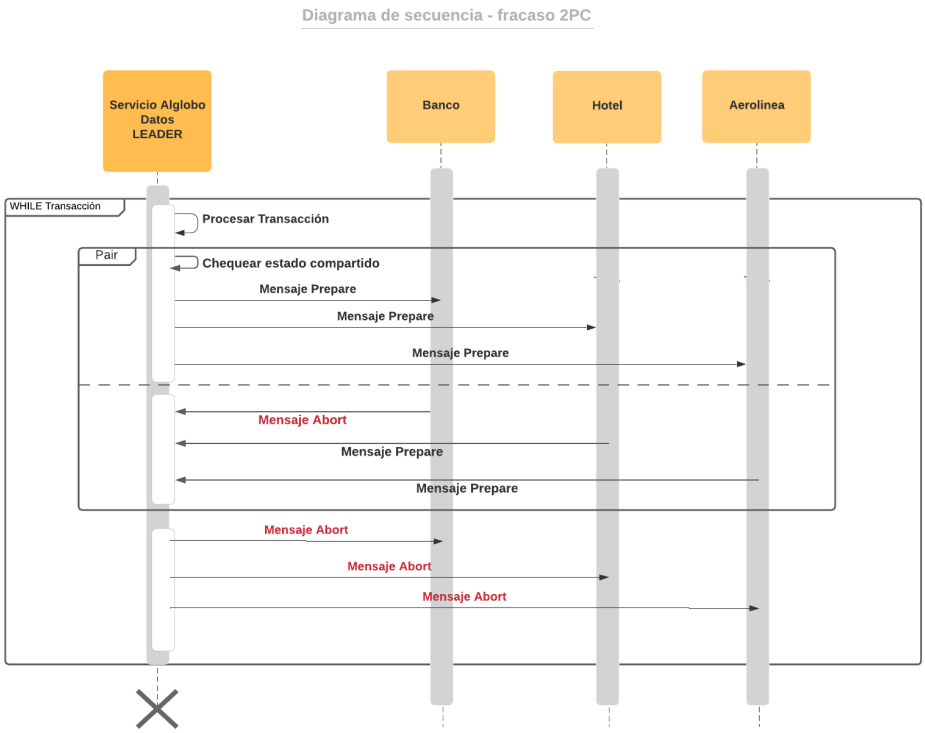


Figura 3: Diagrama de secuencia Two Phase Commit - Failure.

## 5.1. Caso especial: Banco

Para modelar la *reserva, consumo y liberación de recursos*, se implementó especialmente en la entidad **Banco** un modelo muy simple de cuentas bancarias, con dinero disponible, las cuales se leen de un archivo *accounts_ csv*.

De esta manera, cada transacción contiene un *cbu* (identificador de cuenta), así como un costo de aerolínea y hotel asociados. Cuando se hace el *Prepare* (y no se decide por abortar aleatoriamente), se reservan"los recursos, restándolo de la cuenta bancaria correspondiente. Si posteriormente se hace el *Commit*, se confirma ese consumo de dinero. En cambio, si se hace luego un *Abort*, se restablece el dinero que se había restado previamente.

Un detalle a mencionar es que una transacción no falla si no hay saldo suficiente (simplemente lo loguea en consola), sino que el banco hace "préstamos"(o las cuentas pueden tener saldo negativo, según el punto de vista). El modelado de dinero se utilizó para visualizar el objetivo del algoritmo transaccional *Two Phase Commit*, vislumbrando la importancia de la administración de los recursos.

# 6. Conclusiones

A lo largo del TP se trato de mantener el foco en los algoritmos vistos en clase, buscando entender la manera mas eficiente de implementarlos para la problemática dada. En toda situación planteada se analizaron conceptos como transaccionalidad y respuesta ante fallas, logrando así que el sistema continué funcionando ante cualquier eventualidad. También se sistematizo el entorno de desarrollo virtualizando las instancias y aislando la comunicación con una red interna. El equipo se encuentra satisfecho con el trabajo realizado.