Trabajo Práctico 2 - YPF Ruta

Programación Concurrente

108397 -	**Alejo Ordoñez**	github.com/alejoordonez02
105666 -	**Francisco Pereyra**	github.com/fapereyra
107863 -	**Lorenzo Minervino**	github.com/lminervino18
103376 -	**Alejandro Paff**	github.com/AlePaff

2c2025

Índice

Introducción

En este trabajo se presenta **YPF Ruta**, un sistema distribuido que permite a las empresas centralizar el pago y el control del gasto de combustible de su flota de vehículos.

Cada empresa dispone de una cuenta principal y de tarjetas asociadas a los distintos conductores. Cuando un vehículo necesita cargar combustible en cualquiera de las más de 1600 estaciones distribuidas en el país, el conductor utiliza su tarjeta para autorizar la operación. El sistema registra cada consumo y, posteriormente, factura a la empresa el monto total acumulado en todas sus tarjetas durante el período de facturación.

Panorama general del sistema

En esta sección se describe, a grandes rasgos, cómo se organiza **YPF Ruta**: qué actores intervienen, cómo se conectan entre sí y qué problemas busca resolver el sistema a nivel de negocio y de infraestructura distribuida.

Contexto de uso

YPF Ruta modela un escenario en el que empresas con flotas de vehículos cargan combustible en una red amplia de estaciones de servicio. Cada conductor utiliza una tarjeta asociada a la cuenta de su empresa para autorizar la carga, mientras que el sistema central:

- valida los límites de consumo de la tarjeta y de la cuenta,
- registra cada operación de cobro,
- permite consultas de consumo y facturación posterior.

Desde el punto de vista técnico, el sistema asume un entorno distribuido con múltiples nodos comunicados por red, posibles fallas de nodos, desconexiones temporales y estaciones que pueden funcionar momentáneamente sin conectividad directa al clúster de consenso.

Objetivos y alcance del sistema

El objetivo principal de YPF Ruta es proporcionar una infraestructura distribuida que permita:

- Centralizar el control del gasto de combustible de una flota mediante cuentas y tarjetas.
- Replicar el estado de cuentas y tarjetas entre varios nodos (líder y réplicas) manteniendo consistencia mediante un log replicado.
- Tolerar la caída del líder mediante una elección automática de nuevo líder (algoritmo Bully).
- Soportar políticas de cobro en escenarios de desconexión (modo OFFLINE) y reconciliar los consumos una vez recuperada la conectividad.
- Ofrecer una interfaz de administración (CLI) para limitar cuentas y tarjetas, consultar consumos y disparar procesos de facturación.

El sistema se implementa como un prototipo académico: el almacenamiento es en memoria, la lógica de negocio se modela mediante actores y la comunicación entre nodos se realiza sobre TCP con un protocolo de aplicación propio.

Server distribuido

En esta sección se describe la arquitectura interna del servidor de **YPF Ruta** como sistema distribuido. El foco está en el clúster de consenso (líder y réplicas), en el flujo de las operaciones de negocio y en cómo se mantiene un estado consistente entre nodos a través de un log replicado.

Clúster de consenso (líder y réplicas)

El servidor está organizado alrededor de un *clúster de consenso* formado por un nodo **líder** y uno o más nodos **réplica**. Todos ellos mantienen una copia lógica del mismo estado de negocio: cuentas, tarjetas, límites y consumos. Ese estado residente en memoria se implementa mediante un modelo de actores (ActorRouter, AccountActor, CardActor), aislando la lógica de negocio de la lógica de comunicación distribuida.

Desde el punto de vista del clúster:

- El **líder** es el único nodo que decide el *orden global* de las operaciones y coordina el commit. Cada vez que recibe una nueva operación (por ejemplo, un `Charge` generado por una estación o un `LimitAccount` solicitado por un administrador) la registra en su log local y desencadena el proceso de replicación hacia las réplicas.
- Las **réplicas** reciben del líder las entradas del log y las aplican en el mismo orden, manteniendo su propio sistema de actores sincronizado con el del líder. No toman decisiones de orden ni de commit por sí mismas: siguen la secuencia que les envía el líder.

El flujo simplificado para una operación es el siguiente (esquema inspirado en Raft):

1. Una operación de alto nivel (`Operation`) llega a algún nodo del sistema (líder, réplica o estación que proxyea) y termina siendo entregada al líder en forma de mensaje `Request`.
2. El líder asigna un nuevo identificador de operación (`op_id`), la guarda en su estructura de `PendingOperation` y crea una entrada de log asociada.
3. El líder envía a cada réplica un mensaje `Log { op_id, op }` a través de la capa de red (abstracción `Connection`).
4. Cada réplica, al recibir un `Log`, almacena la operación en su propio log y ejecuta, a través de su `Database` (ActorRouter + actores), la operación inmediatamente anterior. Una vez que la ejecución termina, la réplica responde al líder con un mensaje `Ack { op_id }`.
5. El líder va contabilizando los `Ack` recibidos. Cuando una operación tiene *ack* de la mayoría de las réplicas, se considera *committed*. En ese momento el líder ejecuta la operación en su propio sistema de actores y traduce el resultado a una respuesta:
    - si el origen era una estación, genera un `NodeToStationMsg::ChargeResult` hacia la Station;
    - si el origen era un cliente TCP (CLI administrativo), envía un `Message::Response` con el `OperationResult` correspondiente.

De esta manera, el log replicado define un orden total de las operaciones y garantiza que tanto el líder como las réplicas apliquen exactamente la misma secuencia sobre su estado en memoria. La consistencia del clúster se logra sin compartir memoria entre nodos: toda la coordinación se hace mediante mensajes (`Request`, `Log`, `Ack`, `Response`) sobre TCP.

Estaciones y clientes administrativos

Aunque el clúster de consenso está formado únicamente por el líder y las réplicas, **no todas las estaciones de servicio forman parte de ese clúster**. En YPF Ruta distinguimos dos tipos de clientes del servidor:

- Estaciones que no cumplen rol en el clúster de consenso (nodos "externos" al consenso).
- Clientes administrativos (CLI) utilizados por las empresas para gestionar sus cuentas.

Ambos tipos de cliente generan operaciones de alto nivel (`Operation`) que, directa o indirectamente, terminan llegando al líder.

Estaciones de servicio

Cada estación cuenta con un simulador de surtidores (`Station`) que:

- lee comandos desde stdin (uno por operación de cobro),
- los traduce a mensajes de alto nivel (`StationToNodeMsg::ChargeRequest`),
- y espera la respuesta correspondiente (`NodeToStationMsg::ChargeResult`).

Según dónde esté corriendo la estación:

- Si se ejecuta en el mismo proceso que un **nodo líder** o una **réplica**, la `Station` se conecta internamente a ese nodo y éste actúa como "frente" de la estación frente al clúster de consenso.
- Si la estación está completamente **afuera del clúster**, se conecta por TCP a algún nodo (típicamente el líder) y ese nodo toma el rol de entrada para sus operaciones.

En cualquier caso, la estación no necesita conocer la topología interna del clúster ni cómo se implementa la replicación: únicamente envía requests de cobro y recibe resultados admitido/denegado con información opcional de error (`VerifyError`).

Clientes administrativos (CLI)

El cliente administrativo se implementa como un binario independiente (`client`) que ofrece un CLI para interactuar con el sistema. A través de este cliente es posible:

- limitar el monto disponible en una cuenta (`Operation::LimitAccount`),
- limitar el monto disponible en una tarjeta (`Operation::LimitCard`),
- consultar el consumo de una cuenta (`Operation::AccountQuery`),
- iniciar procesos de facturación (`Operation::Bill`).

El CLI serializa estas operaciones a un formato binario propio y las envía por TCP al servidor. Desde la perspectiva del clúster, un comando del CLI es simplemente otra `Operation` que ingresa al líder mediante un mensaje `Request`, sigue el mismo flujo de replicación y commit que un cobro

originado en una estación, y termina en una respuesta `OperationResult` que el cliente muestra por stdout.

Tolerancia a fallas y elección de líder (Bully)

El clúster de consenso está pensado para seguir funcionando incluso si algunos nodos dejan de responder, en particular el líder. Mientras al menos una réplica y la mayoría de los nodos sigan activos, el sistema puede elegir un nuevo líder y continuar procesando operaciones.

Cuando se detecta la caída del líder, una de las réplicas inicia una **elección de líder** usando el algoritmo Bully. La idea básica es:

- Cada nodo del clúster tiene un identificador único (ID numérico).
- El nodo que detecta la falla se postula como candidato y envía mensajes de *elección* a los nodos con ID más alto que el suyo.
- Si ningún nodo con ID mayor responde, el candidato se proclama nuevo líder y anuncia su rol al resto del clúster.
- Si algún nodo con ID mayor responde, ese nodo "toma la posta" y continúa el proceso de elección, hasta que finalmente el nodo con ID más alto disponible se convierta en líder.

De esta forma, el liderazgo siempre recae en el nodo "más fuerte" (ID más alto) que siga activo. Para las estaciones y los clientes administrativos, este proceso es casi transparente: pueden seguir enviando operaciones a los nodos del clúster, que se encargan de redirigirlas internamente hacia el líder vigente.

Clientes

En esta sección se describen los dos tipos principales de clientes que interactúan con **YPF Ruta**: las estaciones de servicio (a través de sus surtidores) y los usuarios administrativos de las empresas, que operan mediante un cliente de línea de comandos (CLI).

Estaciones de servicio y surtidores

Cada estación se modela como una terminal que agrupa varios surtidores. En el prototipo, esta terminal está representada por el componente `Station`, que:

- lee comandos ingresados por el operador (uno por cada intento de carga),
- los traduce a solicitudes de cobro de alto nivel hacia el nodo con el que está conectada,
- y muestra en pantalla el resultado de la operación (aprobada o rechazada, con el motivo).

Desde la perspectiva de la estación, el flujo típico es:

1. El operador ingresa los datos de la operación (surtidor, cuenta, tarjeta, monto).
2. La estación envía una solicitud de cobro al sistema.
3. El sistema responde indicando si la operación fue aceptada o no, y por qué (por ejemplo, límite de tarjeta o de cuenta excedido).

El detalle de cómo se replica esa operación dentro del clúster (líder, réplicas, log, ACKs) queda oculto detrás de esta interfaz simple de "solicitud de cobro / resultado".

Cliente administrativo (CLI) para empresas

Las empresas que utilizan YPF Ruta cuentan además con un cliente administrativo en modo texto (CLI), pensado para tareas de gestión sobre sus cuentas. A través de este cliente es posible:

- establecer o modificar el límite global de una cuenta,
- fijar límites individuales para cada tarjeta asociada,
- consultar el consumo acumulado de una cuenta y su desglose por tarjeta,
- iniciar procesos de facturación sobre una cuenta.

Cada comando del CLI se traduce internamente en una `Operation` (por ejemplo, `LimitAccount`, `LimitCard`, `AccountQuery` o `Bill`) que se envía al servidor y sigue el mismo flujo de consenso que las operaciones de cobro. El usuario administrativo, sin embargo, sólo ve una interfaz simple de consulta y actualización de parámetros de su cuenta.

Implementación del nodo

Cada proceso del servidor (líder o réplica) se modela como un **nodo** que se comunica simultáneamente con tres "mundos" distintos:

- otros nodos del clúster (consenso y replicación),
- la estación local (surtidores simulados),
- la base de datos lógica implementada con actores.

Estos comportamientos se abstraen en el trait `Node`, que es implementado por `Leader` y `Replica`.

Visión general y bucle principal del nodo

El corazón de la implementación es el bucle principal definido en `Node::run`. Este bucle asincrónico se ejecuta mientras el nodo está vivo y resuelve, mediante un `tokio::select!`, tres tipos de eventos:

- mensajes que llegan desde otros nodos a través de la capa de red (`Connection`),
- mensajes que llegan desde la estación local (`StationToNodeMsg`),
- eventos emitidos por el mundo de actores (`ActorEvent`) a través de `Database`.

Según el tipo de evento, el nodo delega en los métodos del trait `Node`:

- `handle_node_msg` para mensajes de consenso y replicación (`Request`, `Log`, `Ack`, `Election`, etc.),
- `handle_station_msg` para solicitudes de cobro y cambios de modo ONLINE/OFFLINE,
- `handle_actor_event` para resultados de operaciones de negocio.

De esta forma, la lógica específica de líder o réplica se concentra en la implementación del trait, mientras que el bucle principal es común a todos los roles.

Comunicación con otros nodos (Connection / Message)

La comunicación entre nodos se encapsula en la abstracción `Connection`, que ofrece una interfaz uniforme para:

- aceptar conexiones entrantes y establecer conexiones salientes,

- enviar mensajes tipados (`Message`) a una dirección (`SocketAddr`),
- recibir mensajes desde la red de forma asincrónica.

El enum `Message` representa todos los mensajes de "mundo nodo":

- mensajes de operaciones (`Request`, `Response`, `Log`, `Ack`),
- mensajes de elección de líder (`Election`, `ElectionOk`, `Coordinator`),
- mensajes de membresía del clúster (`Join`, `ClusterView`, `ClusterUpdate`).

El método `handle_node_msg` del trait `Node` actúa como *dispatcher*: desempaqueta el `Message` recibido y llama a los handlers específicos (`handle_request`, `handle_log`, `handle_ack`, `handle_election`, etc.) que implementan la semántica de líder o réplica.

Comunicación con la estación (Station, StationToNodeMsg, NodeToStationMsg)

La interacción con los surtidores de una estación se modela mediante el tipo `Station`, que corre en una tarea de fondo y se comunica con el nodo por canales asincrónicos:

- `StationToNodeMsg`: mensajes que la estación envía al nodo.
- `NodeToStationMsg`: mensajes que el nodo envía de vuelta a la estación.

Los mensajes principales son:

- `StationToNodeMsg::ChargeRequest` para solicitar un cobro (cuenta, tarjeta, monto).
- `StationToNodeMsg::DisconnectNode` y `ConnectNode` para cambiar el modo ONLINE/OFFLINE.
- `NodeToStationMsg::ChargeResult` para devolver el resultado final de una operación de cobro, incluyendo si fue permitida o no y un posible `VerifyError`.
- `NodeToStationMsg::Debug` para enviar mensajes informativos al operador.

El método `handle_station_msg` del trait `Node` encapsula la política de qué hacer ante cada mensaje: en el caso de un `ChargeRequest`, construye una `Operation::Charge` y la inyecta en el flujo normal del clúster (pasando por el líder y la replicación).

Comunicación con la base de datos lógica (Database, ActorEvent)

La **base de datos lógica** del sistema está implementada como un conjunto de actores y se encapsula detrás del tipo `Database`. Desde el punto de vista del nodo, `Database` ofrece dos operaciones:

- `send(DatabaseCmd)`: para enviar comandos de alto nivel (por ahora, `Execute { op_id, operation }`).
- `recv()`: para recibir eventos producidos por el mundo de actores (`ActorEvent`).

El actor principal es el `ActorRouter`, que coordina:

- actores de cuenta (`AccountActor`) responsables de los límites y consumos a nivel cuenta,
- actores de tarjeta (`CardActor`) responsables de los límites y consumos por tarjeta.

Cuando el líder decide *commit* de una operación, en lugar de aplicar la lógica de negocio directamente, envía un `DatabaseCmd::Execute` a `Database`. El resultado llega luego como un `ActorEvent::OperationResult`, que el nodo traduce a:

- una respuesta a la estación (`NodeToStationMsg::ChargeResult`), o
- una respuesta al cliente administrativo (`Message::Response` con un `OperationResult`).

Esta separación permite que el nodo se concentre en coordinación distribuida y tolerancia a fallas, mientras que la consistencia y verificación de las operaciones de negocio se resuelven dentro del modelo de actores.

Modelo de actores de la base de datos

La base de datos lógica de **YPF Ruta** se modela con el patrón de **actores**: cada entidad de negocio (cuenta, tarjeta) es un actor con estado propio, que solo se modifica a través de mensajes. El nodo nunca accede a ese estado directamente: envía operaciones de alto nivel y recibe un resultado tipado.

ActorRouter

`ActorRouter` es la puerta de entrada al mundo de actores. Desde el punto de vista del nodo:

- recibe una operación de negocio (por ejemplo, `Execute(op_id, Operation)`),
- se asegura de tener creados los actores necesarios (cuentas y tarjetas),
- coordina el intercambio de mensajes entre ellos,
- y, cuando todo termina, emite un único resultado `OperationResult(op_id, Operation, Resultado)`.

En síntesis, toma un pedido de alto nivel, lo descompone en mensajes internos y vuelve a concentrar todas esas interacciones en una respuesta única para el nodo.

AccountActor

Cada `AccountActor` encapsula el estado de una cuenta:

- límite global de la cuenta,
- consumo acumulado,
- y, cuando hace falta, estado temporal para consultas con detalle por tarjeta.

Conceptualmente maneja tres tipos de mensajes:

- `ApplyCharge(amount, from_offline_station)`
- `ApplyAccountLimit(new_limit)`
- `AccountQueryStep(card_id, consumed)`

Con ellos:

- valida y aplica cargos a nivel cuenta (respetando el límite global),
- valida y actualiza el límite global,
- y agrega la información que le van reportando las tarjetas para construir la respuesta de una consulta de cuenta.

CardActor

Cada `CardActor` representa una tarjeta individual:

- límite por tarjeta,
- consumo acumulado,
- y una cola de tareas para procesar sus operaciones en orden.

Recibe, de forma conceptual, mensajes del estilo:

- `ExecuteCharge(amount, from_offline_station)`
- `ExecuteLimitChange(new_limit)`
- `ReportStateForAccountQuery()`

Con estos mensajes:

- verifica el límite de tarjeta antes de un cargo,
- delega en la cuenta la verificación del límite global,
- y responde con su consumo actual cuando se arma una consulta de cuenta.

Mensajes, operaciones y resultados (Operation, OperationResult)

Para desacoplar el nodo del modelo de actores se define un conjunto acotado de operaciones y resultados de alto nivel:

`Operation` = - `Charge(account_id, card_id, amount, from_offline_station)` - `LimitAccount(account_id, new_limit)` - `LimitCard(account_id, card_id, new_limit)` - `AccountQuery(account_id)` - `Bill(account_id, period)`

`OperationResult` = - `ChargeResult(Ok | Failed(VerifyError))` - `LimitAccountResult(Ok | Failed(VerifyError))` - `LimitCardResult(Ok | Failed(VerifyError))` - `AccountQueryResult(account_id, total_spent, per_card_spent)`

El flujo completo es:

1. El nodo recibe una `Operation` y decide cuándo se *commitea* según el consenso.
2. Una vez decidido, envía `Execute(op_id, Operation)` al modelo de actores.
3. Los actores de cuenta y tarjeta procesan la operación únicamente mediante mensajes.
4. `ActorRouter` devuelve un `OperationResult`, que el nodo traduce en la respuesta a la estación o al cliente administrativo.

Protocolo de comunicación

Protocolo de aplicación (formato de mensajes y serialización)

Los distintos procesos de **YPF Ruta** se comunican mediante un protocolo de aplicación binario propio. Cada mensaje comienza con un byte de tipo que identifica la variante (`Request`, `Log`, `Ack`, `Election`, etc.), seguido por los campos específicos de ese tipo codificados en binario (enteros, `f32`, banderas, etc.).

Este formato se usa de manera uniforme tanto entre nodos del clúster de consenso como entre clientes (estaciones o CLI administrativo) y el nodo al que se conectan. Por encima de estos mensajes de bajo nivel, la lógica del sistema trabaja con operaciones de negocio (`Operation`) y resultados (`OperationResult`), lo que permite separar las decisiones de dominio de los detalles de serialización.

Protocolo de transporte (TCP)

Para el transporte se utiliza **TCP**, que ofrece un canal fiable, orientado a conexión y con entrega ordenada de los bytes. Estas propiedades son fundamentales en un sistema de cobros, donde cada mensaje representa una operación económica y cada entrada de log debe aplicarse en el mismo orden en todos los nodos del clúster. Además, el cierre o error en la conexión TCP se usa como señal de caída de un nodo o cliente, disparando los mecanismos de tolerancia a fallas cuando corresponde.

Política de cobro en estaciones sin conexión

Nodo fuera del clúster

Cuando una estación que **no forma parte del clúster de consenso** pierde conexión con la red, el sistema central no puede verificar límites ni saldos. En ese escenario la decisión de aceptar o rechazar el cobro queda íntegramente del lado de la estación: puede operar en modo "confío y anoto localmente" o en modo "no autorizo sin conexión", pero cualquier política que adopte será necesariamente local y no consistente con el resto del sistema hasta que se recupere la conectividad.

Nodo réplica / líder en modo OFFLINE

Si quien pierde conectividad es un nodo del clúster (líder o réplica), puede entrar explícitamente en modo **OFFLINE**. En este modo el nodo deja de participar del consenso, pero sigue atendiendo a sus surtidores: cada cobro se acepta inmediatamente y la operación se registra en una cola de "cargos offline" marcada como tal (`from_offline_station = true`). La idea es priorizar la continuidad de servicio en la estación, posponiendo la verificación global de límites para cuando el nodo vuelva a estar en línea.

Reconciliación al recuperar la conexión

Al volver a modo **ONLINE**, el nodo reproduce secuencialmente todas las operaciones encoladas contra la base de datos lógica y las replica al clúster de consenso. Como esas operaciones están etiquetadas como provenientes de una estación offline, se aplican directamente sobre el estado (sin volver a bloquear al cliente) para reconstruir un historial consistente. Si durante la desconexión se eligió un nuevo líder, las actualizaciones pendientes se envían a ese líder vigente, de modo que el clúster converge nuevamente a un único estado acordado.

Cambios respecto a la primera entrega

Durante la implementación del sistema se realizaron cambios significativos respecto al diseño inicial presentado en la primera entrega. Estos cambios surgieron de una mejor comprensión de los

requisitos del sistema y de las herramientas de concurrencia distribuida disponibles.

1 - Cambio de arquitectura: de clústeres por tarjeta a consenso centralizado

El cambio fundamental fue pasar de un sistema con **múltiples clústeres descentralizados por tarjeta** a un **clúster de consenso centralizado con log replicado**. Esta decisión simplificó enormemente la implementación, permitió demostrar de forma más clara las herramientas de concurrencia distribuida (elección de líder, consenso) y resultó en un sistema más robusto y fácil de razonar

Diseño inicial: El sistema se planteó con tres tipos de clústeres: Clúster de surtidores en cada estación, clúster de nodos suscriptores por tarjeta (con líder de tarjeta) y clúster de cuenta que coordinaba líderes de tarjetas Esta arquitectura buscaba optimizar la comunicación mediante localidad geográfica, donde cada tarjeta tenía su propio clúster de nodos suscriptores que se replicaban la información, con un mecanismo de TTL (Time-To-Live) para desuscribirse de tarjetas no utilizadas.

Implementación final: Se adoptó un modelo de **consenso centralizado** con: - Un nodo **líder** único que coordina todas las operaciones - Nodos **réplica** que mantienen copias sincronizadas del estado - Un **log replicado** inspirado en Raft para garantizar consistencia

Razones del cambio: 1. **Simplicidad:** El modelo de consenso centralizado es más simple de implementar correctamente y razonar sobre su comportamiento. 2. **Consistencia fuerte:** El log replicado garantiza que todas las operaciones se apliquen en el mismo orden en todos los nodos, evitando inconsistencias complejas. Asi como tecnicas de consenso y tolerancia a fallas 3. **Scope del prototipo:** Para un sistema académico con 1600 estaciones simuladas, la optimización por localidad geográfica agregaba complejidad innecesaria.

2 - Modelo de actores: de actores distribuidos a actores locales

Diseño inicial: Se plantearon tres tipos de actores distribuidos: Actor **Suscriptor**, Actor **Líder Tarjeta** y Actor **Cuenta**

Los actores se comunicaban entre nodos mediante propagación viral de mensajes.

Implementación final: Los actores quedaron como entidades **locales** dentro de cada nodo y que además permiten - `ActorRouter`: punto de entrada y coordinador - `AccountActor`: maneja el estado de una cuenta (límite y consumo) - `CardActor`: maneja el estado de una tarjeta (límite y consumo)

Razones del cambio: 1. **Separación de responsabilidades:** Los actores se enfocan exclusivamente en la lógica de negocio (validar límites, acumular consumos), mientras que el consenso distribuido lo maneja el clúster de nodos. 2. **Simplicidad del modelo:** Los actores solo se comunican mediante mensajes dentro del mismo proceso, eliminando la complejidad de comunicación inter-nodo a nivel de actores.

3 - Eliminación del mecanismo de TTL y suscripciones dinámicas

Diseño inicial: Los nodos se suscribían dinámicamente a tarjetas según uso geográfico, con un mecanismo de TTL para desuscribirse automáticamente.

Implementación final: Se eliminó completamente este mecanismo. Todos los nodos del clúster replican todo el estado desde el inicio.

4 - Protocolo de consenso: log replicado con mayoría de ACKs

Diseño inicial: No se especificaba un protocolo formal de consenso entre nodos del clúster.

Implementación final: Se implementó un protocolo de log replicado simplificado: 1. El líder asigna un `op_id` secuencial a cada operación 2. El líder envía `Log { op_id, operation }` a todas las réplicas 3. Cada réplica ejecuta la operación **anterior** y responde con `Ack { op_id }` 4. El líder espera ACKs de la mayoría antes de considerar la operación *committed* 5. El líder ejecuta la operación en su propio sistema de actores y responde al cliente

5 - Manejo de desconexiones: modo OFFLINE con reconciliación

Diseño inicial: Se mencionaba que las estaciones sin conexión podían realizar cobros que se encolaban para posterior sincronización.

Implementación final: Se formalizó el concepto de **modo OFFLINE**: - Cualquier nodo (líder o réplica) puede entrar explícitamente en modo OFFLINE - En modo OFFLINE, los cobros se aceptan inmediatamente y se encolan con flag `from_offline_station = true` - Al volver a modo ONLINE, se reproducen todas las operaciones encoladas contra el clúster - Las operaciones offline se aplican sin re-validar límites (ya fueron consumidas)

6 - Protocolo de serialización binario

Diseño inicial: Se especificaba un protocolo con 1 byte de tipo de mensaje y campos específicos por tipo.

Implementación final: Se mantuvo la idea general pero se refinó: - Byte inicial identifica el tipo de `Message` o `Operation` - Campos serializados en orden fijo (u32, u64, f32, etc.) - Sin compresión ni optimizaciones avanzadas (simplicidad sobre eficiencia)

Conclusiones

El desarrollo de **YPF Ruta** permitió diseñar e implementar un sistema distribuido capaz de centralizar el control de gasto de combustible de una flota, manteniendo al mismo tiempo un nivel alto de disponibilidad en las estaciones. La combinación de un clúster de consenso (líder + réplicas) con un modelo de actores para la base de datos lógica separa con claridad las preocupaciones: por un lado, la replicación y el acuerdo sobre el orden de las operaciones; por otro, la consistencia de las reglas de negocio sobre cuentas y tarjetas.

El rol del **líder** como punto de compromiso de las operaciones, junto con las **réplicas** que mantienen copias sincronizadas del estado, aporta tolerancia a fallas a nivel de nodo. La elección de líder basada en el algoritmo **Bully** permite recuperar un coordinador válido cuando se detecta la caída del nodo actual, evitando que el sistema quede indefinidamente sin un responsable de avanzar el log de operaciones. Desde el punto de vista de los clientes (estaciones y CLI administrativo) esta transición es transparente: basta con reenviar la operación al nodo que responde efectivamente.

El uso del **modelo de actores** para representar cuentas y tarjetas introduce un esquema de concurrencia que evita compartir memoria mutable entre hilos y nodos. Todas las modificaciones de estado se expresan como mensajes de alto nivel (`Operation`) y sus correspondientes resultados (`OperationResult`), lo que simplifica tanto el razonamiento sobre la lógica de negocio como la integración con el protocolo de replicación. La política explícita para el **modo OFFLINE** en estaciones o nodos del clúster, junto con la reconciliación posterior, permite balancear continuidad de servicio y consistencia eventual del sistema.

En conjunto, la arquitectura propuesta muestra que es posible construir un servicio de cobros distribuido que combina: replicación y consenso sobre operaciones, encapsulamiento de la lógica de negocio en actores y un protocolo de comunicación binario simple pero suficiente para las necesidades del dominio. A partir de esta base se pueden explorar extensiones naturales, como nuevas operaciones administrativas, métricas de uso por estación, o políticas más sofisticadas de manejo de riesgo en m