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 desarrolla **YPF Ruta**, un sistema que permite a las empresas centralizar el pago y el control de gasto de combustilble para su flota de vehículos.
Las empresas tienen una cuenta principal y tarjetas asociadas para cada uno de los conductores de sus vehículos. Cuando un vehículo necesita cargar en cualquiera de las 1600 estaciones distribuídas alrededor del país, puede utilizar dicha tarjeta para autorizar la carga; siendo luego facturado mensualmente el monto total de todas las tarjetas a la compañía.

Aplicaciones

Server

El servidor consiste de un sistema distribuido en el que la información con respecto a las cuentas y tarjetas se encuentra centralizada en un clúster compuesto de un *nodo líder* y *nodos réplica*. La entidad por default para un nodo del sistema es *estación*; esto es, todos los nodos cumplen con el rol de ser una estación y además pueden ser líder o réplica.
A nivel estación, los *surtidores* que residen en ella se intercomunican para mantener la funcionalidad de la estación, y lograr así una abstracción de los surtidores a nivel sistema global.

Cliente

El único cliente (fuera del servidor de YPF) es el **administrador**. El administrador puede

- Limitar los montos disponibles en su cuenta.
- Limitar los montos disponibles en las tarjetas de la cuenta.
- Consultar los saldos de las cuentas.
- Consultar los saldos de las tarjetas de la cuenta.
- Realizar la facturación de la cuenta.

Arquitectura del servidor

Como ya se mencionó, el servidor está implementado de manera distribuida. El foco principal del diseño de la arquitectura está en reducir la cantidad de mensajes entre nodos que tienen viajar en la red.

Dado que la información se encuentra centralizada en el *clúster de consenso*, se vuelve necesario que las estaciones consulten el estado de la información de la cuenta a la cuál pertenece la tarjeta que quiere realizar el pago en ellas.
Para ésto, las estaciones conocen incialmente *quién* es el nodo líder. Cualquier consulta que precisen hacer se la envían al mismo. Si el líder dejara de funcionar, se ejecutaría entonces un algoritmo de elección de líder como *bully-algorithm* para elegir un nuevo de entre las réplicas, para luego actualizar a todas las estaciones con el resultado de la elección.

De esta manera todos los nodos necesitan un único socket para comunicarse con el nodo líder, salvo éste último que necesita tantos sockets como estaciones existan además de él. Esto ocurre a nivel lógico, ya que estaciones poco concurridas no necesitan estar constantemente conectadas con el nodo líder, por lo que el mismo tiene la posibilidad de mantener sólo un top N conexiones con el resto de las estaciones. Cuando pasa un tiempo sin que se envíen mensajes del sistema, la conexión se cierra para ahorrar recursos.

Tipos de clúster

A continuación se explican más en profundidad cada uno de los tipos de clúster que se mencionaron.

Clúster de surtidores.

Los surtidores en una estaicón se conectan directamente al servidor que ejecuta la funcionalidad de nodo en el sistema global-los surtidores no son computadoras, son hardware que envía I/O al servidor de la estación-, por lo que la concurrencia en éste clúster es a nivel memoria. Para prevenir las race conditions que surgen del acceso concurrente de lecto escritura a memoria, se utiliza el modelo de actores.

Clúster de consenso

El clúster de consenso está conformado por un único nodo líder y N réplicas de la información que éste contiene. Si el líder deja de funcionar, las réplicas lo detectan y inician la re-elección mediante un *bully-algorithm*.
Para evitar desincronización en casos falla de alguno de los nodos del clúster de consenso, se utiliza algoritmo de sincronización de transacciones *two-phase commit*.

Clúster de estaciones (*sistema global*)

El **clúster de estaciones** se refiere a todos los nodos que se ejecutan en las estaciones de YPF. Todos las estaciones deben cumplir con éste mínimo rol: poder realizar el cobro de cargarle nafta a un conductor de YPF Ruta.

Paseo por varios casos de uso

1. *Un conductor usa su tarjeta por primera vez en el surtidor de una estación.*

A nivel estación, quien recibe la responsabilidad de realizar el cobro a una tarjeta es un surtidor. Como la tarjeta no se encuentra aún cargada en el sistema, cuando el nodo estación envía la consulta sobre la disponibilidad de saldo de la misma (o de su cuenta), el nodo líder genera el registro de la tarjeta, así como también de la cuenta a la que esta pertenece si no existiera aún; y envía el nuevo registro a los nodos réplica.
Si la estación es el nodo líder entonces el checkeo se realiza en memoria en vez de mediante un paquete de red. Los nodos réplica no tienen ningún comportamiento especial fuera del flujo que se sigue al registro de la tarjeta-envían la consulta por red al líder.

3. *Un conductor usa su tarjeta en una nueva estación nueva, habiéndola usado en otras.*

Si un conductor usa su tarjeta en una nueva estación, es decir, en una estación en la que todavía no la había usado, entonces la estación no va a contar con el registro de la tarjeta y por tanto propagará la consulta como en el caso 1. Ésta vez si va a recibir una respuesta de una de los nodos que estén suscriptos a la tarjeta, por lo que

1. gestiona el cobro como en 2,
2. envía el mensaje de *suscripción*,
3. invita a sus nodos cercanos,
4. y actualiza a la lista de nodos suscriptos por el cobro realizado.

Node failure recovery

Hasta ahora sólo consideramos los casos felices del funcionamiento del sistema, pero en la realidad los nodos pueden fallar. A continuación detallamos lo que pasaría en caso de que cada uno de los distintos tipos de nodos falle, a partir de la siguiente configuración arbitraria del sistema:

Política de cobro en estaciones sin conexión

Si una estación se encuentra sin conexión, no hay nada que hacer si se trata de un nodo que está fuera del clúster de consenso. Se puede o bien realizar el cobro o no.
Tampoco hay mucho más que hacer cuando cuando se trata de un nodo réplica o líder, más que tomar una política de asumir que la información que se tiene está actualizada o no. En el primer caso, el nodo réplica o líder, revisa el saldo restante de la tarjeta (y de la cuenta a la que pertenece) y realiza el cobro en base a esa información-si no hay saldo suficiente niega la operación. En el segundo caso, la operación se lleva a cabo sin revisar el registro de la tarjeta (ni el de la cuenta a la que pertenece).

Sin importar el nodo o la política que se le aplique, en caso de que el cobro finalmente se efectúe, la actualización del registro de la tarjeta debe ser encolada para poder ser enviada al líder del clúster de consenso una vez recuperada la conexión.

Si fuera el nodo líder el que perdió la conexión, entonces cuando la recuperase, ya se habría elegido

a otro y por tanto sería a ese nuevo líder al que se enviarían las actualizaciones encoladas si así las hubiera.

Flujo de las consultas de los clientes

Modelo de Actores

Actores

Mensajes del sistema

Mensaje	Descripción
Cobrar	Solicitud de cobro iniciada por un surtidor o reenviada entre nodos
RespuestaCobro	Confirmación o rechazo del cobro (por límite de tarjeta o de cuenta)
Actualización	Propagada a todos las réplicas del clúster de consenso después de un cobro exitoso

Representación en pseudocódigo Rust

```rust
enum ErrorCobro {
    // ...
}

// ======== Mensajes ========
enum Mensaje {
    // flujo principal
    Cobrar {
        // suscriptor -> líder tarjeta & líder tarjeta -> cuenta
        transaction_id: TransactionID,
        tarjeta: TarjetaID,
        monto: f64,
        origen: ActorID,
        destino: ActorID,
    },
    RespuestaCobro {
        // líder tarjeta -> suscriptor & cuenta -> líder tarjeta
        transaction_id: TransactionID,
        ok: bool,
        razon: ErrorCobro,
        monto: f64,
    },
    Actualizacion {
        // actualizaciones de límites de tarjetas/cuentas
    },
}

// ======== Estructura de los registros ========
struct RegistroTarjeta {
    tarjeta: TarjetaID,
    cuenta: CuentaID,
    saldo_usado: f64,
    limite_tarjeta: f64,
    ttl: u8, // tiempo de vida de la suscripción
    lider_id: NodoID, // líder de la tarjeta
}

// ======== Suscriptor ========
struct Suscriptor {
    id: ActorID,
    tarjeta: RegistroTarjeta,
    pendientes: HashMap<TransactionID, f64>,
}

impl Suscriptor {
    // entrada del cobro en un surtidor a un suscriptor que contiene el nodo
    fn handle_cobrar(&mut self, transaction_id: TransactionID, tarjeta: TarjetaID,
```

```
        monto: f64) {
        tarjeta.ttl = INITIAL_TTL;
        if monto + tarjeta.gastado > tarjeta.limite {
            // no puedo cobrar
        }

        self.lider
            .send(Mensaje::Cobrar::new(transaction_id, tarjeta, monto));
    }

    fn handle_respuesta_cobro(
        &mut self,
        transaction_id: TransactionID,
        ok: bool,
        razon: String,
        monto: f64,
    ) {
        if ok {
            // se puede cargar nafta
            return;
        }

        // no se puede cargar nafta
    }

    // actualizar el registro de la tarjeta
    fn handle_actualizacion(&mut self, tarjeta: TarjetaID, monto: f64) {
        self.tarjeta.saldo_usado += monto;
        if transaction.origin != self {
            self.tarjeta.ttl -= 1;
            if self.tarjeta.ttl == 0 {
                self.tarjetas.remove(&tarjeta);
            }
        }
    }
}

// ======== Líder ========
struct Lider {
    id: ActorID,
    tarjeta: TarjetaID,
    cuenta: ActorID,
    suscriptores: Vec<ActorID>,
    pendientes: HashMap<TransactionID, (ActorID, f64)>,
}

impl Lider {
    // recibe Cobrar desde un suscriptor y lo reenvía a la cuenta
```

```
    fn handle_cobrar(
        &mut self,
        transaction_id: TransactionID,
        tarjeta: TarjetaID,
        monto: f64,
        origen: NodoID,
    ) {
        self.cuenta
            .send(Mensaje::Cobrar::new(transaction_id, tarjeta, monto));
    }

    // recibe la decisión de la cuenta
    fn handle_respuesta_cobro(
        &mut self,
        transaction_id: TransactionID,
        ok: bool,
        razon: String,
        monto: f64,
    ) {
        // responder al suscriptor que inició el flujo
        origen_suscriptor.enviar(
            origen_suscriptor,
            Mensaje::RespuestaCobro {
                transaction_id,
                ok,
                razon,
                monto: monto_guardado,
            },
        );

        // actualizar al resto de suscriptores del clúster
        for s in &self.suscriptores {
            s.enviar(
                Mensaje::Actualizacion::new(
                    tarjeta: self.tarjeta.clone(),
                    delta: monto_guardado,
                )
            );
        }
    }
}

// ======== Cuenta ========
struct Cuenta {
    id: NodoID,
    cuenta: CuentaID,
    limite: f64,
    consumo_total: f64,
```

```rust
}

impl Cuenta {
    // valida el límite global de la cuenta y decide el cobro
    fn handle_cobrar(
        &mut self,
        transaction_id: TransactionID,
        _tarjeta: TarjetaID,
        monto: f64,
        origen: ActorID, // líder
    ) {
        if self.consumo_total + monto >= self.limite {
            origen.enviar(
                Mensaje::RespuestaCobro::new(
                    transaction_id,
                    ok: false,
                    razon: ErrorCobro::LimiteCuenta;
                    monto,
                )
            );

            return;
        }

        self.consumo_total += monto;
        origen.enviar(
            Mensaje::RespuestaCobro::new(
                transaction_id,
                ok: true,
                razon: "".into(),
                monto,
            )
        );
    }
}
```

Protocolo de comunicación

Por tratarse de un sistema distribuido tienen que comunicarse por red. Es por esto que se hace necesario introducir un protocolo de aplicación y el elegir un protocolo de capa de transporte.

Protocolo de capa de aplicación

Si bien los nodos tienen acceso al código de la implementación de las entidades del sistema, no comparten memoria, si no que se comunican enviando mensajes por red, y por tanto se hace necesario introducir un **protocolo** de *serialización* y *deserialización* de las tiras de bytes que se envían.
El protocolo es simple, todos los mensajes tienen 1 byte para el tipo de mensaje (disponibilidad para $2^{1\times8} = 256$ tipos de mensaje distintos), de manera tal que el resto de la deserialización se lleva a cabo según este tipo. Además, cuando un actor quiere comunicarse con otro actor, delega esta comunicación a la abstracción de envío de mensajes entre actores en distintos nodos, por lo tanto todos los mensajes están *wrappeados* en un protocolo de más bajo nivel que contiene la información de la comunicación entre los nodos. Para los mensajes descriptos en la descripción del modelo de autores, se propone la siguiente estructura:

- `Cobrar.` `req_id`: ID de un actor, `tarjeta_id`: ID de una tarjeta, `monto`: doble precisión, `origen_id`: ID de un actor.
- `RespuestaCobro.` `req_id`: ID de un actor, `ok`: booleano, `razon`: enum de tipos de error, `monto`: doble precisión.
- `RegistroCobro.` `req_id`: ID de un actor, `registro`: *registro de tarjeta*.
- `Actualización.` `tarjeta`: ID de una tarjeta, `delta`: doble precisión.

El *registro de tarjeta* es `RegistroTarjeta:` `tarjeta`: ID de la tarjeta, `cuenta`: ID de la cuenta, `saldo_usado`: doble precisión, `limite_tarjeta`: doble precisión, `ttl`: entero positivo, `lider_id`: ID de un actor.

Por último, los campos están definidos de la siguiente manera:

- **ID de una tarjeta.** No sabemos cuántas tarjetas hay, por lo que usamos 4 bytes para representar sus IDs ($2^{4\times8}$, más de 4 mil millones de IDs distintos).
- **ID de un actor.** Debería haber más capacidad para actores que tarjetas en el sistema, por lo que se utilizan para definirlos 5 bytes.
- **Doble precisión.** Usamos el estándar 754 de la IEEE de doble precisión para todos los números que representan montos y fracciones de tiempo. Son 8 bytes: 1 bit de signo, 11 bits para el exponente y el resto de los 52 bits para la mantisa. En Rust esto es un `f64`.
- **Entero positivo.** Si sólo se usase para el TTL entonces bastaría con tener un byte para este campo.
- **Enum de tipos de error.** Un sólo byte para poder representar hasta 256 tipos de erorres distintos. Luego durante la deserialización debería traducirse el tipo a un mensaje legible por el usuario (si es que no se trata de un error del sistema que pueda ser manejable por el mismo).

Protocolo de capa de transporte

En el sistema **YPF Ruta** se utiliza el protocolo **TCP (Transmission Control Protocol)** para la comunicación local entre los distintos nodos del sistema.

TCP garantiza la **entrega confiable y ordenada** de los mensajes, propiedad esencial en un entorno donde cada operación representa una transacción económica. Además provee la detección de interrupciones de comunicación, que es muy útil a la hora de detectar fallas de nodos.