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 existen tres tipos diferentes de clústers de nodos:

- *Surtidores* en una estación.
- *Nodos suscriptos a una tarjeta.*
- *Nodos líderes de tarjetas* que forman una *cuenta.*

Entidades que participan:

- **Surtidores.** Los surtidores corresponden a las máquinas interconectadas de manera *local* en una estación.
- **Estaciones/Nodos.** Los nodos representan estaciones de YPF. Dentro de una estación, uno de los surtidores tiene la responsabilidad de llevar a cabo la función del nodo en el sitema global.

Hay tres tipos de nodos:

- **Suscriptor (tarjeta).** Los nodos suscriptores mantienen informados a sus pares (otros nodos suscriptos a la misma tarjeta) sobre las actualizaciones al registro de la tarjetas a la que suscriben. Un nodo puede estar suscripto a varias tarjetas.
- **Líder (tarjeta).** Los nodos líder *lideran* un clúster de nodos suscriptores a una tarjeta; ésto es: tienen la responsabilidad de intercomunicar a los nodos del clúster y a su vez de informar sobre actualizaciones de la tarjeta al *nodo cuenta* cuando este así lo solicite. Un nodo líder es también un nodo suscriptor.
- **Cuenta.** Los nodos cuenta se comunican con un nodo líder de cada una de las tarjetas que le pertenecen a la cuenta. Un nodo cuenta **no** puede ser el líder de un clúster de nodos suscriptos a una tarjeta.

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, partiendo de la arquitectura trivial: un grafo completo, con réplicas de la información del sistema en todos los nodos.

Se pueden hacer varias optimizaciones a partir de algunas observaciones del *modelo de negocio* del sistema. Existe localidad con respecto al posicionamiento geográfico de las estaciones; un conductor que aparece en una estación probablemente vuelva a aparecer en estaciones cercanas, y probablemente no aparazca en una estación en la otra punta del país (o al menos no con frecuencia significativa).
Una forma de optimizar la comunicación entre nodos sería entonces tenerlos separados por cuentas: cada nodo tendría una réplica de la información de todas las tarjetas de la cuenta a la que pertenece y sólo debería comunicar a los otros nodos del clúster de la cuenta respecto de las actualizaciones de la misma.
El problema con esto último es que una empresa grande, con muchas tarjetas y muchos conductores a lo largo del país; tendría réplicas innecesarias: un conductor que vive en Salta probablemente no use una estación en Santa Cruz, sin embargo, si uno de sus compañeros de trabajo así lo hace, entonces el registro de su tarjeta estaría replicado en la estación de Santa Cruz.
La solución que se encontró es la de dividir los clústers por tarjeta y no por cuenta. Ahora bien, como también necesitamos centralizar la información de todas las tarjetas pertenecientes a una cuenta, surge la necesidad de los nodos *cuenta*. Para minimizar la comunicación de los nodos cuenta con los nodos de las tarjetas que le pertenecen, el rol de comunicador se centraliza en los nodos *líder tarjeta*.

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 estación están conectados de manera local y se encargan de mantener actualizado al surtidor líder del clúster para que este ejerza la función de nodo estación en el sistema global.

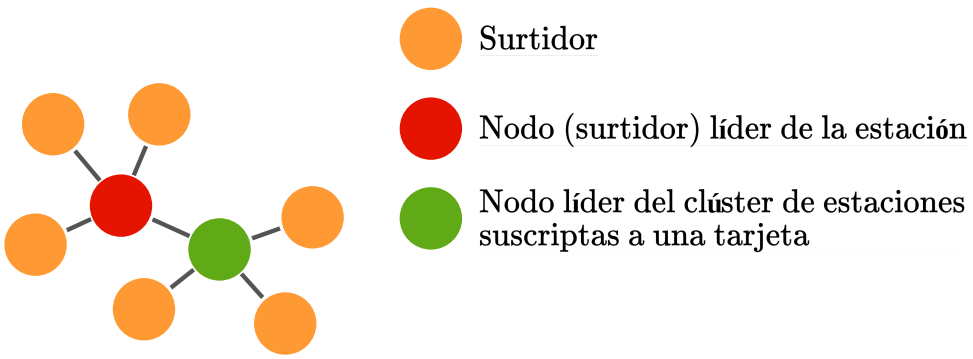

Figura 1: Dos estaciones, con cuatro surtidores cada una.

Clúster de nodos suscriptos a una tarjeta.

Los nodos suscriptos a una tarjeta informan a sus pares de las actualizaciones en los registros de las tarjetas a las que suscriben. Hay un líder del clúster y los *súbditos* se encargan de elegirlo al principio de la ejecución y en caso de que el mismo deje de estar activo.

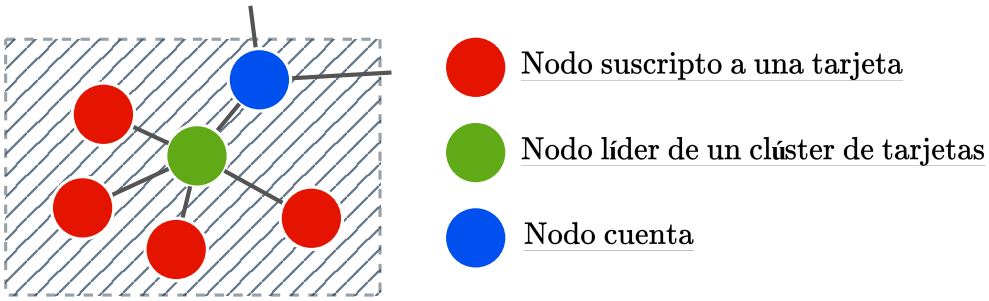

Figura 2: Clúster de nodos suscriptos a una tarjeta.

Clúster de cuenta.

El clúster de nodos líderes de tarjetas tienen su propio líder: el *nodo cuenta*. Dentro de éste clúster se mantiene actualizado al nodo cuenta ante cualquier cambio en alguno de los registros de las tarjetas que conforman la cuenta. Los *súbditos* eligen un líder al principio de la ejecución y en caso de que el mismo deje de estar activo. Las actualizaciones son comunicadas sólo cuando el nodo cuenta así lo solicita.

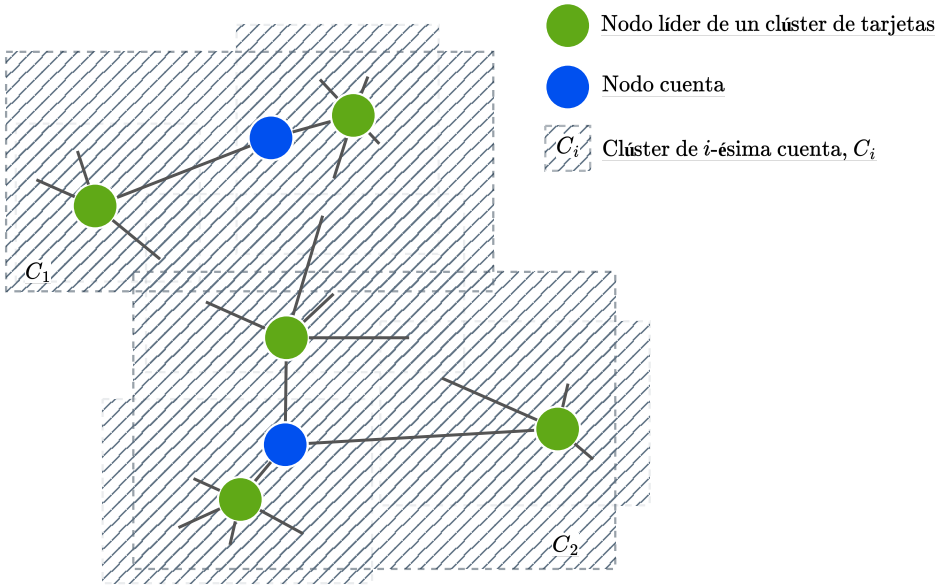

Figura 3: Clúster de cuenta.

Vista de águila

Cabe recalcar que los nodos cuenta (azules) no pueden ser nodos líderes de tarjetas (verdes). Por otro lado, los nodos líder tarjeta (verdes) siempre son suscriptores a la tarjeta que lideran (rojos);

más aún, todos los nodos del sistema cumplen mínimamente con el rol de suscriptor.
En resumen:

- los nodos cuenta y los nodos líder tarjeta ejecutan también la responsabilidad de nodos suscriptores,
- los nodos líder tarjeta son, en particular, suscriptores a la tarjeta que lideran (también pueden estar suscriptos a otras tarjetas)
- y los nodos cuenta no pueden ser nodos líder. Si un nodo líder tarjeta asume la responsabilidad de ser un nodo cuenta, entonces tiene que delegar la responsabilidad de líder tarjeta a otro nodo del clúster de suscriptores a la tarjeta; de donde surge una última regla:
- un clúster de nodos suscriptos a una tarjeta tiene que cumplir con una cantidad mínima. En caso de no hacerlo, se invita a un nodo del sistema a suscribirse a la tarjeta.

Agrupando los niveles de clúster (y obviando los surtidores), la vista general de una posible configuración del sistema se ve de la siguiente forma:

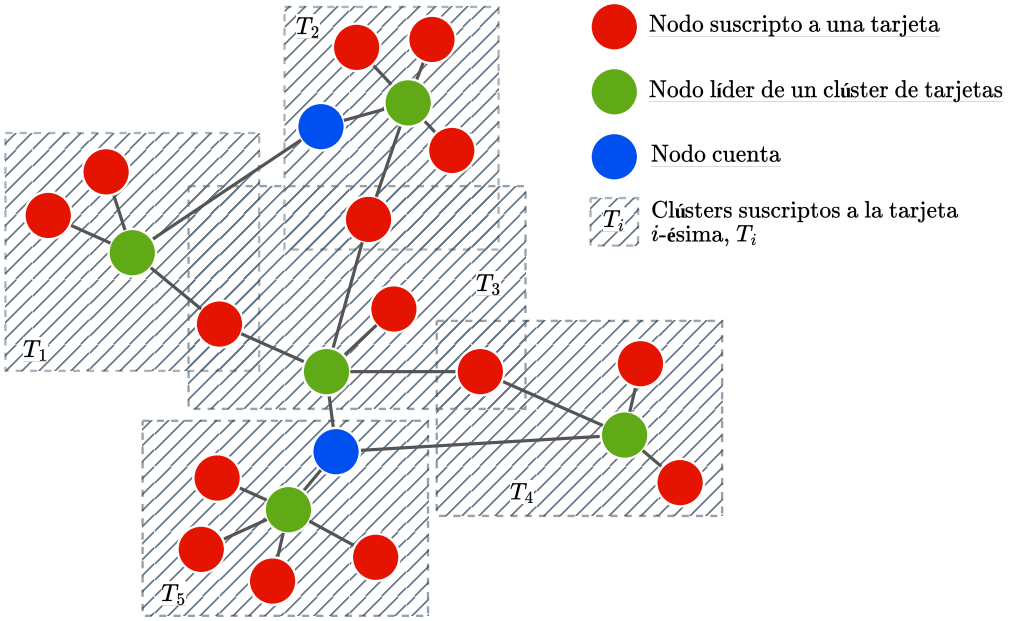

Figura 4: *Overview* del sistema distribuido global.

Paseo por varios casos de uso

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

1. El conductor le da su tarjeta al cajero, que usa la terminal de cobro de la columna del surtidor que usó para cargar nafta. El surtidor necesita saber si el cobro puede o no ser efectuado. Para ello revisa la información de la tarjeta, como no la tiene en guardada, la solicita. El mensaje utilizado para la solicitud es delegado al nodo central de la estación. En este punto ya nos encontramos en el sistema distribuido de estaciones.

2. Una vez que el nodo estación recibe el mensaje con la solicitud de información de la tarjeta, envía el mensaje a sus estaciones vecinas, y así lo hacen estas últimas, propagando el mensaje como un *virus*. El mensaje que se propaga contiene, además de la solicitud en sí misma, las direcciones a las que ya se propagó; para evitar demasiados mensajes redundantes.
   Como esta es la primera vez que la tarjeta es utilizada, ningún nodo va a contestar con su información y por lo tanto el nodo de la estación original genera el registro de la tarjeta.
   En caso de que el mensaje llegue a un nodo cuenta al que le pertenece la tarjeta, el mismo puede rápidamente contestar si la tarjeta ya existe o no.

3. Una vez generado el registro, se deben tener un mínimo de nodos suscriptos a la misma, un nodo cuenta líder y un nodo cuenta generado para la tarjeta. Como ningún nodo cuenta contestó, y no ningún otro nodo tenía la tarjeta, se generan ambos. Además se invitan a la lista de suscripción al top N nodos más cercanos para replicar en ellos la información del registro de la misma, y también porque el sistema no acepta un nodo que sea cuenta y líder tarjeta en simultáneo.

4. Con todas las condiciones del sistema distribuido en orden, la estación procede a realizar el cobro para luego actualizar a los suscriptores de la tarjeta (que acaban de generarse).

2. *Un conductor usa su tarjeta en el surtidor de una estación a la que frecuenta.*

Si un conductor utiliza su tarjeta en una estación a la que va con frecuencia, entonces ésta estación ya tiene cargado el registro de la tarjeta. Aún así, se necesita saber si a la cuenta le queda monto para realizar el cobro, para esto se procede de la siguiente manera:

1. El surtidor envía la consulta de saldo de cuenta al nodo líder de la estación.
2. El nodo líder de la estación envía la consulta de saldo de cuenta al nodo líder tarjeta.
3. El nodo líder tarjeta envía la consulta al nodo cuenta.
4. El nodo cuenta consulta las actualizaciones de los nodos líder del resto de tarjetas, computa la respuesta y se la envía al nodo líder tarjeta que le hizo la consulta.

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.

Time-to-leave (TTL)

Supongamos que un conductor utiliza siempre su tarjeta en las estaciones cercanas a su casa en Córdoba. Si el conductor se va, de manera espontánea, de viaje a Formosa (por trabajo, si no no usaría la tarjeta de la empresa…), entonces probablemente utilice varias estaciones entre Córdoba y Formosa. Cuando vuelva de de su jornada laboral (o de sus vacaciones si no hizo un buen uso de la tarjeta), no volvería a usar su tarjeta en las estaciones en las que la usó para viajar a Formosa.
Sería un desperdicio de recursos—mínimos en memoria, pero sí significativos para la comunicación en la red—tener un nodo suscrito a la lista de una tarjeta si éste no fuera a volver a ser utilizado.
Por esto se introduce el campo **TTL** en los registros de las tarjetas. Si un nodo es actualizado de manera *externa*, es decir, se actualiza la información de un registro de una de sus tarjetas sin que la tarjeta haya efectuado la carga en esa estación; un número mayor a TTL veces, entonces se elimina de la lista de suscripción de la tarjeta. De esta forma, evitamos que con el paso del tiempo el sistema gaste recursos actualizando a estaciones a las que no les debería importar el registro de una tarjeta.

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:

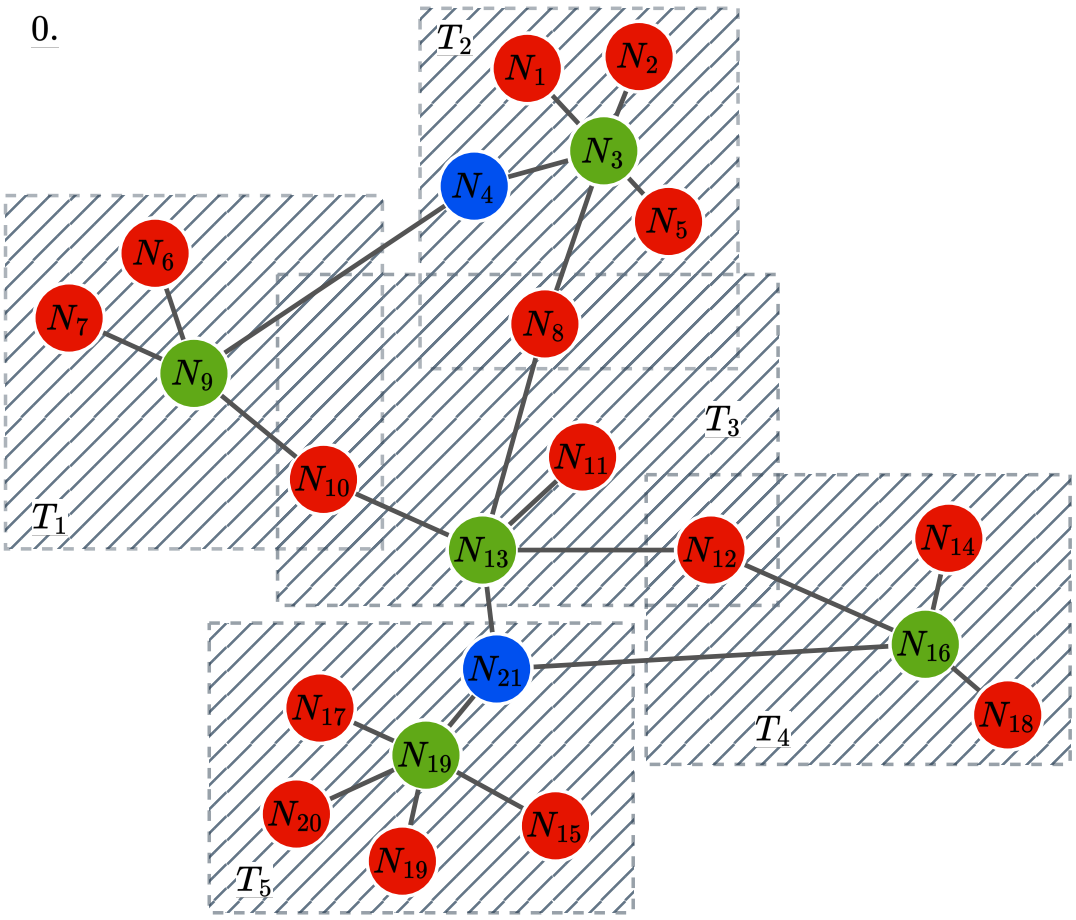

Figura 5: Estado inicial del sistema. Dos cuentas, una con las tarjetas T_1 y T_2 y la otra con T_3, T_4 y T_5.

Se cae N_1: nodo suscriptor.

Que se caiga un nodo suscriptor no representa un problema demasiado grande. En este caso la estación va a tener que guardarse las actualizaciones a la tarjeta, sin poder realizar las consultas de suficiencia de saldo en las mismas o en sus cuentas. No hay nada más que hacer puesto que la única responsabilidad del nodo suscriptor es comunicar al nodo líder y no hay nunca posibilidad de que esto así ocurra.
Cuando el nodo vuelve a la vida, tiene que preguntar quién es el leader, enviarle sus actualizaciones de la tarjeta a la que el clúster suscribe para que este actualice al resto de nodos en el clúster y al nodo recuperado, a este último con la agregación de las actualizaciones que acaba de enviar y las que se efectuaron durante su baja.

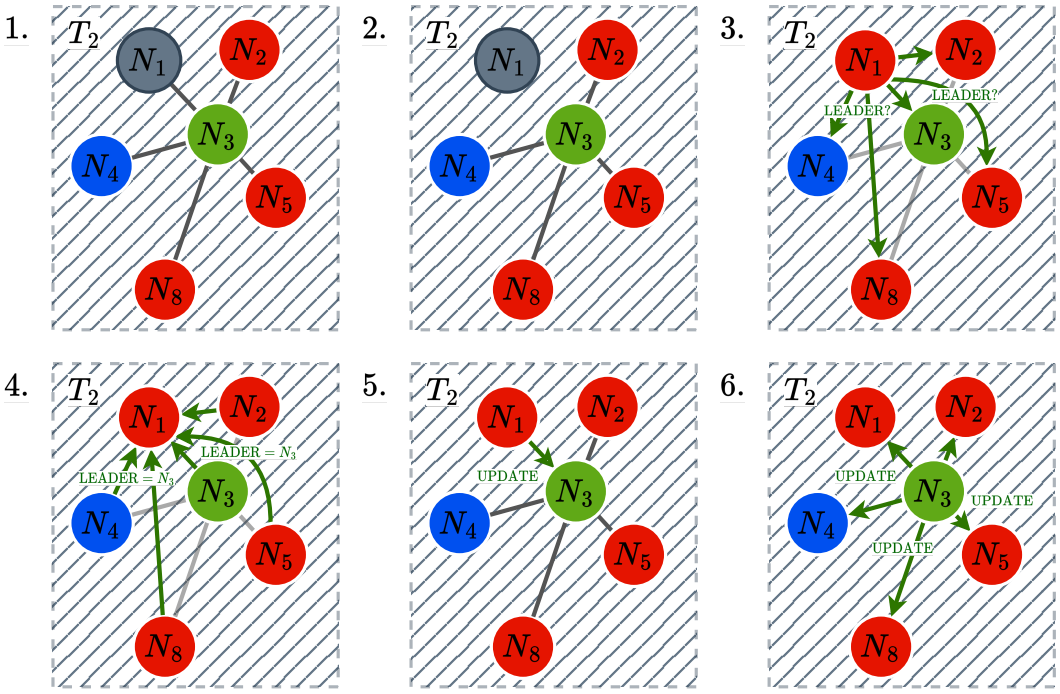

Figura 6: *Recovery* de la falla en N_1.

Se cae N_{13}: nodo líder tarjeta.

Que se caiga un nodo líder tarjeta representa un mayor problema ya que su responsabilidad es la de centralizar la información generada por un clúster sobre una tarjeta y estar disponible para cuando el nodo cuenta al que pertenece la tarjeta consulte la información de la misma. En este caso se usa el algoritmo de elección de líder *Bully* y se comunica al nodo cuenta sobre el líder elegido.
En caso de que sea el nodo cuenta quien se entera de la baja del nodo líder, simplemente envía un mensaje de elección de líder, sin participar de la elección, y recibir el líder elegido al final de la misma.

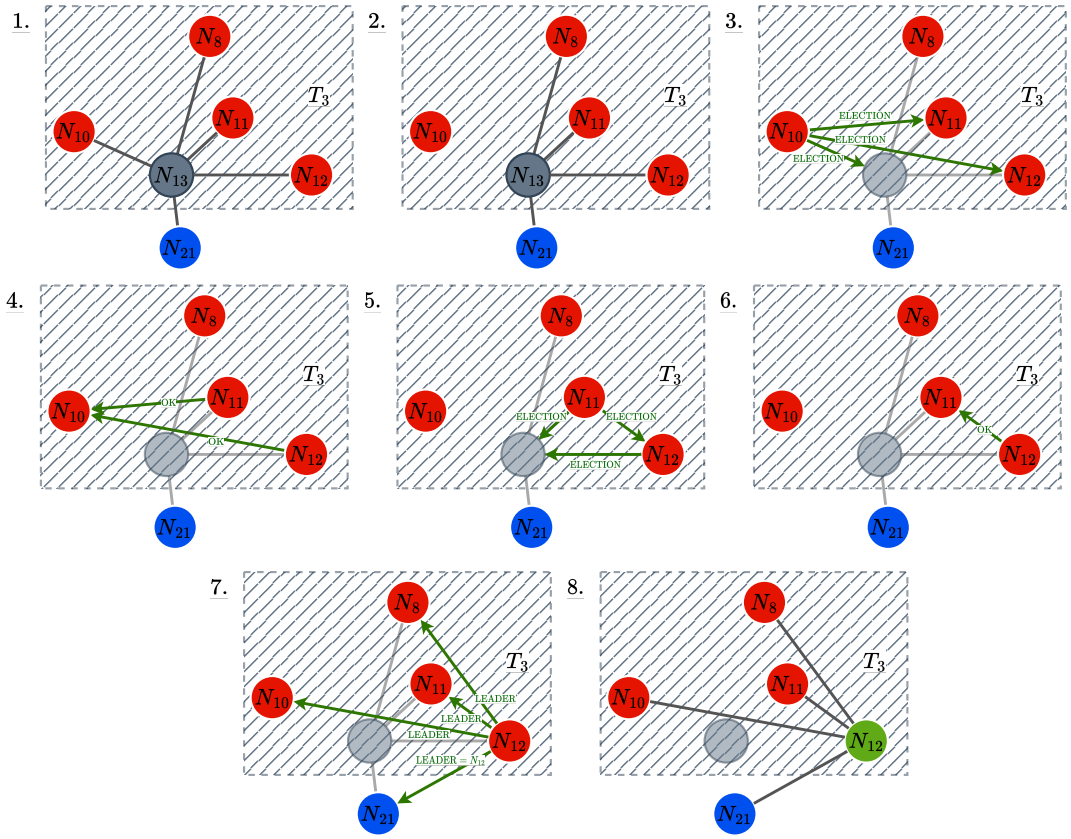

Figura 7: *Recovery* de la falla en N_{13}.

Se cae N_{22}: nodo cuenta.

Este es el caso más complicado, ya que el nodo cuenta es el tipo de nodo con mayor responsabilidad del sistema. La dinámica de recovery de este caso es muy similar a la de cuando se cae un nodo líder tarjeta, sumando una re-elección del nodo líder tarjeta ya que las responsabilidades líder tarjeta y cuenta no son compatibles.

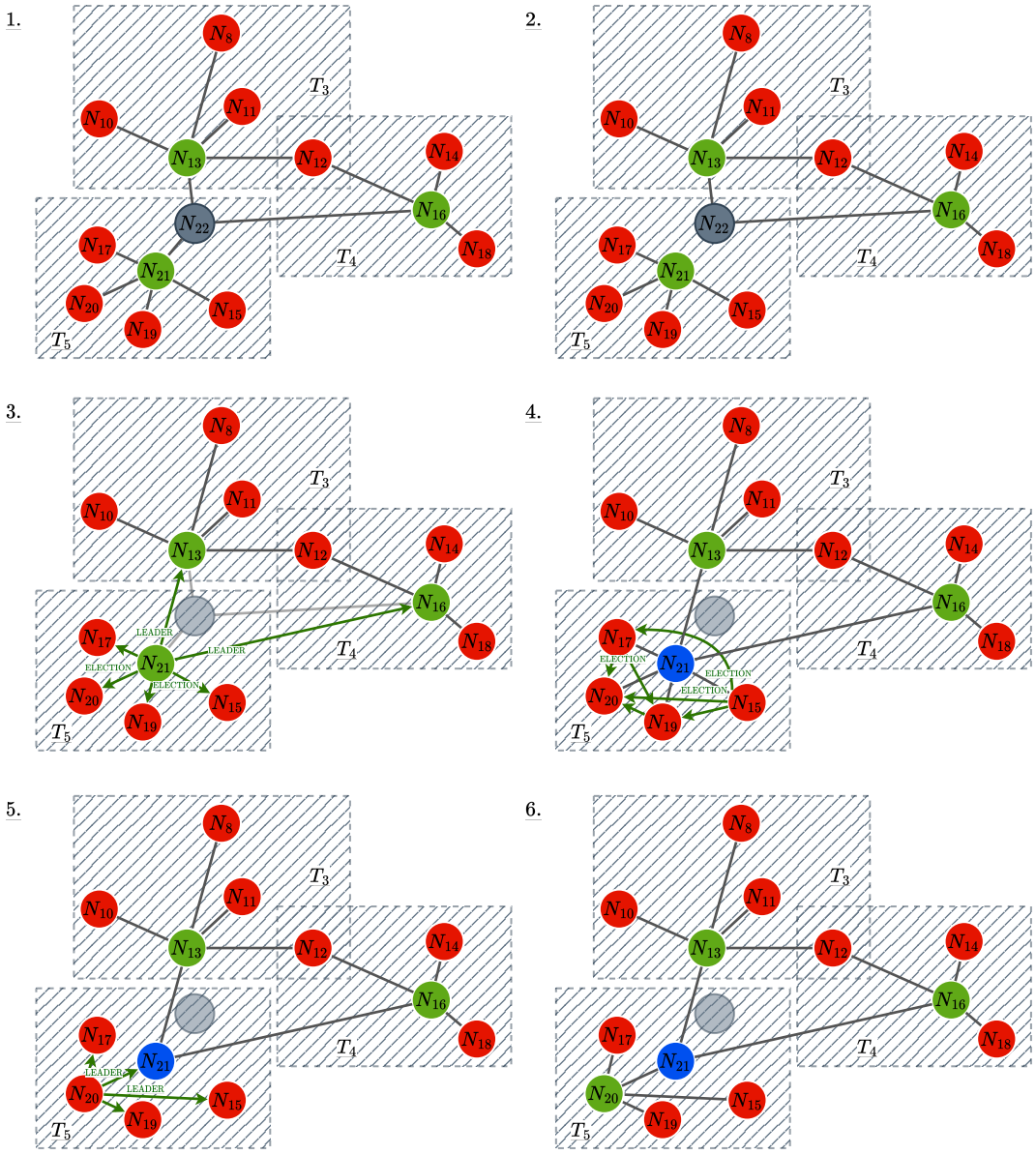

Figura 8: *Recovery* de la falla en N_{22}. En este diagrama se obvia el algoritmo *bully* para elegir nodo líder del clúster suscripto a la tarjeta T_5 puesto que ya se mostró en mayor detalle en el caso anterior. (4.) Existen optimizaciones como hacer que N_{21} mande un sólo mensaje de ELECTION a los nodos del clúster, pero en sí la idea es logar que los nodos elijan a un nuevo líder, ya que los nodos cuenta no pueden ser nodos líder tarjeta. (5.) Notar además que es N_{21} quien se encarga de poner al clúster de suscriptores T_5 en modo elección, para no quedar elegido, siendo que es el de mayor ID, puede simplemente no contestar, o contestar con mensaje del tipo CANNOT.

Flujo de las consultas de los clientes

Cuando un **administrador** hace una consulta o impone un nuevo monto límite, ya sea de su cuenta principal o de una sus tarjetas, el mensaje de la consulta se direcciona a al nodo más cercano geográficamente. El nodo del server que recibe la consulta, checkea si tiene o no el registro y si no lo tiene propaga el mensaje de la misma forma en la que lo haría si se tratara de una consulta de registro entre nodos. Cuando llega el registro, el nodo que recibió la consulta del cliente, se suscribe al mismo y contesta al administrador, de esta manera ninguna estación está sobrecargada con peticiones de los clientes y las posteriores consultas se responden más rápido.

Modelo de Actores

El sistema se modela siguiendo el **paradigma de actores distribuidos**, donde cada proceso representa una entidad que se comunica mediante el envío de mensajes.
Cada actor mantiene su propio estado interno y procesa mensajes de forma asíncrona, garantizando independencia y resiliencia ante fallos.

Actores

- **Nodo Surtidor**: ejecuta en la red local de una estación y envía solicitudes de cobro al nodo estación correspondiente.

- **Nodo Estación Suscriptor**: mantiene registros locales de tarjetas que se usaron en su zona. Si no conoce una tarjeta, propaga el intento de cobro a sus estaciones vecinas.
  Además, valida el **límite de la tarjeta** antes de delegar el cobro al nodo líder.

- **Nodo Estación Líder**: lidera un clúster de suscriptores de una tarjeta. Recibe cobros de los suscriptores y los reenvía al nodo cuenta para su validación global. Luego distribuye las actualizaciones a los nodos suscriptores.

- **Nodo Cuenta**: centraliza el control del saldo total de la cuenta principal. Valida los límites globales de la empresa y confirma o rechaza las operaciones.

Por diseño, **cada cuenta tiene un único nodo cuenta** y cada tarjeta tiene un único **nodo líder**, que actúa como intermediario entre el nodo cuenta y los nodos suscriptores.

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)
Registro	Información completa de una tarjeta, enviada cuando un nodo la conoce
Suscripción	Petición para ser agregado a la lista de suscriptores de una tarjeta
Actualización	Propagada a todos los suscriptores después de un cobro exitoso

Representación en pseudocódigo Rust

```
// ======== Tipos auxiliares ========
type NodoID = String;
type TarjetaID = String;
type CuentaID = String;
type ReqID = u64;

// ======== Definición de mensajes ========
// Todos los mensajes que forman parte del flujo de cobro llevan req_id para
    correlación.

enum Mensaje {
    // Flujo principal
    Cobrar { req_id: ReqID, tarjeta: TarjetaID, monto: f64, origen: NodoID },
    RespuestaCobro { req_id: ReqID, ok: bool, razon: String, monto: f64 },

    // Descubrimiento y suscripción
    Registro { req_id: ReqID, tarjeta: TarjetaID, registro: RegistroTarjeta },
    Suscripcion { tarjeta: TarjetaID, nodo: NodoID },

    // Replicación eventual
    Actualizacion { tarjeta: TarjetaID, delta: f64 },
}

// ======== Estructura de los registros ========

#[derive(Clone)]
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 (para enviar Suscripcion/Cobrar)
}

// ======== Nodo Surtidor ========

struct NodoSurtidor {
    id: NodoID,
    estacion: NodoID,
    seq: ReqID, // generador local de req_id
}

impl NodoSurtidor {
    fn cobrar(&mut self, tarjeta: TarjetaID, monto: f64) {
        let req = self.nuevo_req();
        enviar(self.estacion.clone(), Mensaje::Cobrar { req_id: req, tarjeta, monto,
```

```
            origen: self.id.clone() });
    }

    fn handle_respuesta(&self, req_id: ReqID, ok: bool, razon: String, monto: f64) {
        if ok {
            mostrar(format!("[{}] Cobro ${:.2} realizado con éxito", req_id, monto));
        } else {
            mostrar(format!("[{}] Cobro rechazado: {}", req_id, razon));
        }
    }

    fn nuevo_req(&mut self) -> ReqID { self.seq += 1; self.seq }
}

// ======== Nodo Estación Suscriptor ========

struct NodoSuscriptor {
    id: NodoID,
    tarjetas: HashMap<TarjetaID, RegistroTarjeta>,
    vecinos: Vec<NodoID>,
    // req_id -> a quién debo devolver el resultado final (surtidor)
    pendientes: HashMap<ReqID, NodoID>,
    // req_id -> marca de que ya pedí/propagué y estoy esperando Registro (caso 3)
    esperando_registro: HashSet<ReqID>,
}

impl NodoSuscriptor {
    // Entrada principal del flujo desde surtidor o vecinos
    fn handle_cobrar(&mut self, req_id: ReqID, tarjeta: TarjetaID, monto: f64, origen:
        NodoID) {
        // Si el origen es un surtidor, recordar a quién responder.
        self.pendientes.entry(req_id).or_insert(origen.clone());

        if let Some(r) = self.tarjetas.get(&tarjeta) {
            // --- Validación del límite de tarjeta ---
            if r.saldo_usado + monto > r.limite_tarjeta {
                let reply_to = self.pendientes.remove(&req_id).unwrap_or(origen);
                enviar(reply_to, Mensaje::RespuestaCobro {
                    req_id, ok: false, razon: "Límite de tarjeta alcanzado".into(), monto
                });
                return;
            }

            // Enviar al líder de esta tarjeta para validación global en NodoCuenta
            enviar(r.lider_id.clone(), Mensaje::Cobrar {
                req_id, tarjeta: tarjeta.clone(), monto, origen: self.id.clone()
            });
        } else {
```

```
93          // Tarjeta desconocida: dos caminos posibles (caso 1 o caso 3)
94
95          // (A) Propagar el intento de cobro a vecinos (caso 3: alguno conoce y
                responde Registro)
96          if !self.esperando_registro.contains(&req_id) {
97              self.esperando_registro.insert(req_id);
98              for v in &self.vecinos {
99                  enviar(v.clone(), Mensaje::Cobrar {
100                     req_id, tarjeta: tarjeta.clone(), monto, origen: self.id.clone()
101                 });
102             }
103             // (B) Si tras un timeout no llega Registro, asumir PRIMER USO (caso 1)
104             // y auto-generar registro + líder + cuenta locales.
105             // *Aquí lo modelamos como una función que se dispara luego de un
                    timeout.*
106             programar_timeout(self.id.clone(), req_id, tarjeta.clone(), monto);
107         } else {
108             // Este suscriptor fue alcanzado por una propagación vecina.
109             // Si ÉL conoce la tarjeta (caso 3 lado vecino), responde con REGISTRO al
                    origen.
110             if let Some(reg) = self.tarjetas.get(&tarjeta) {
111                 enviar(origen, Mensaje::Registro {
112                     req_id, tarjeta: tarjeta.clone(), registro: reg.clone()
113                 });
114             } else {
115                 // No conoce tampoco →continúa la propagación (evitar loops con
                        TTL/visitados si se desea).
116                 for v in &self.vecinos {
117                     if v != &origen {
118                         enviar(v.clone(), Mensaje::Cobrar {
119                             req_id, tarjeta: tarjeta.clone(), monto, origen:
                                    self.id.clone()
120                         });
121                     }
122                 }
123             }
124         }
125     }
126 }
127
128 // Llega un REGISTRO desde un vecino (caso 3)
129 fn handle_registro(&mut self, req_id: ReqID, tarjeta: TarjetaID, registro:
        RegistroTarjeta) {
130     self.esperando_registro.remove(&req_id);
131     self.tarjetas.insert(tarjeta.clone(), registro.clone());
132     // Suscribirse al líder de la tarjeta
133     enviar(registro.lider_id.clone(), Mensaje::Suscripcion { tarjeta:
            tarjeta.clone(), nodo: self.id.clone() });
```

```
134
135         // Validar límite de tarjeta y continuar el flujo normal hacia el líder
136         let monto_y_reply = self.pendientes.get(&req_id).cloned();
137         if let Some(_reply_to) = monto_y_reply {
138             // No guardamos monto acá, confiamos en que nos llega por Actualizacion y
                RespuestaCobro.
139         }
140
141         if let Some(r) = self.tarjetas.get(&tarjeta) {
142             enviar(r.lider_id.clone(), Mensaje::Cobrar {
143                 req_id, tarjeta: tarjeta.clone(), monto: /* monto real del req_id */
                    recuperar_monto(req_id),
144                 origen: self.id.clone()
145             });
146         }
147     }
148
149     // Llega la respuesta final del líder (éxito o rechazo)
150     fn handle_respuesta_cobro(&mut self, req_id: ReqID, ok: bool, razon: String, monto:
        f64) {
151         let reply_to = self.pendientes.remove(&req_id);
152         if let Some(dest) = reply_to {
153             enviar(dest, Mensaje::RespuestaCobro { req_id, ok, razon, monto });
154         }
155     }
156
157     // Replicación eventual: actualización del saldo local y manejo de TTL
158     fn handle_actualizacion(&mut self, tarjeta: TarjetaID, delta: f64) {
159         if let Some(r) = self.tarjetas.get_mut(&tarjeta) {
160             r.saldo_usado += delta;
161             r.ttl = r.ttl.saturating_sub(1);
162             if r.ttl == 0 {
163                 self.tarjetas.remove(&tarjeta);
164             }
165         }
166     }
167 }
168
169 // ======== Nodo Estación Líder ========
170
171 struct NodoLider {
172     id: NodoID,
173     tarjeta: TarjetaID,
174     cuenta: CuentaID,
175     nodo_cuenta: NodoID,
176     suscriptores: Vec<NodoID>,
177     // req_id -> (origen_suscriptor, monto)
178     pendientes: HashMap<ReqID, (NodoID, f64)>,
```

```
}

impl NodoLider {
    // Recibe Cobrar desde un suscriptor y lo reenvía al NodoCuenta
    fn handle_cobrar(&mut self, req_id: ReqID, tarjeta: TarjetaID, monto: f64, origen:
        NodoID) {
        self.pendientes.insert(req_id, (origen.clone(), monto));
        enviar(self.nodo_cuenta.clone(), Mensaje::Cobrar {
            req_id, tarjeta, monto, origen: self.id.clone()
        });
    }

    // Recibe la decisión del NodoCuenta
    fn handle_respuesta_cobro(&mut self, req_id: ReqID, ok: bool, razon: String, monto:
        f64) {
        if let Some((origen_suscriptor, monto_guardado)) =
            self.pendientes.remove(&req_id) {
            if ok {
                // Propagar actualización a todos los suscriptores (incluido quien inició)
                for s in &self.suscriptores {
                    enviar(s.clone(), Mensaje::Actualizacion {
                        tarjeta: self.tarjeta.clone(), delta: monto_guardado
                    });
                }
            }
            // Responder al suscriptor que inició el flujo
            enviar(origen_suscriptor, Mensaje::RespuestaCobro { req_id, ok, razon,
                monto: monto_guardado });
        }
    }

    fn handle_suscripcion(&mut self, tarjeta: TarjetaID, nodo: NodoID) {
        if tarjeta == self.tarjeta && !self.suscriptores.contains(&nodo) {
            self.suscriptores.push(nodo);
        }
    }
}

// ======== Nodo Cuenta ========

struct NodoCuenta {
    id: NodoID,
    cuenta: CuentaID,
    limite: f64,
    consumo_total: f64,
}

impl NodoCuenta {
```

```
223     // Valida el límite global de la cuenta y decide el cobro
224     fn handle_cobrar(&mut self, req_id: ReqID, _tarjeta: TarjetaID, monto: f64, origen:
            NodoID) {
225         if self.consumo_total + monto <= self.limite {
226             self.consumo_total += monto;
227             enviar(origen, Mensaje::RespuestaCobro {
228                 req_id, ok: true, razon: "".into(), monto
229             });
230         } else {
231             enviar(origen, Mensaje::RespuestaCobro {
232                 req_id, ok: false, razon: "Límite de cuenta principal alcanzado".into(),
                        monto
233             });
234         }
235     }
236 }
237
238 // ======== Notas operativas ========
239 // - programar_timeout: si expira y no llegó Registro, se asume "primer uso" y se crean
240 // RegistroTarjeta, NodoLider y NodoCuenta locales (caso 1), y se reintenta Cobrar.
241 // - recuperar_monto(req_id): en una implementación real, el NodoSuscriptor guardaría
        req_id -> monto
242 // en un HashMap para enviar el monto correcto al reenviar tras recibir Registro.
243 // - enviar(): primitiva de envío de mensajes entre actores (TCP).
```

Flujos de mensajes detallados con validaciones

Caso 1: Primer uso de la tarjeta

1. **Surtidor → NodoSuscriptor:** `Cobrar(tarjeta, monto)`

2. **NodoSuscriptor → Vecinos:** `Cobrar(tarjeta, monto)`
   Ningún nodo responde → *primer uso detectado*

3. **NodoSuscriptor crea:** `RegistroTarjeta + NodoLíder + NodoCuenta`

4. **NodoSuscriptor** valida límite de tarjeta *(local = 0, OK)*

5. **NodoSuscriptor → NodoLíder:** `Cobrar(tarjeta, monto)`

6. **NodoLíder → NodoCuenta:** `Cobrar(tarjeta, monto)`

7. **NodoCuenta** valida límite de cuenta:
   Si **OK** → actualiza `consumo_total` y responde `(ok = true)`
   Si **excede** → responde `(ok = false, "Límite de cuenta principal")`

8. **NodoLíder** recibe respuesta:
   Si **OK** → envía `Actualización` a suscriptores *(propagando el cobro)*
   En cualquier caso → responde al **NodoSuscriptor**

9. **NodoSuscriptor** actualiza su registro local y `TTL`

10. **NodoSuscriptor → Surtidor:** `RespuestaCobro(ok)`

Caso 2: Uso en estación frecuente

1. **Surtidor → NodoSuscriptor:** `Cobrar(tarjeta, monto)`

2. **NodoSuscriptor** valida límite de tarjeta:
   Si **excede** → responde `"Límite de tarjeta alcanzado"`
   Si **OK** → **NodoSuscriptor → NodoLíder:** `Cobrar(tarjeta, monto)`

3. **NodoLíder → NodoCuenta:** `Cobrar(tarjeta, monto)`

4. **NodoCuenta** valida límite global:
   Si **excede** → `RespuestaCobro(false)`
   Si **OK** → `RespuestaCobro(true)`

5. **NodoLíder** recibe la respuesta y:
   Si **OK** → `Actualización` a suscriptores *(saldo actualizado)*

6. **NodoLíder → NodoSuscriptor:** `RespuestaCobro(ok)`

7. **NodoSuscriptor** actualiza su registro + `TTL`

8. **NodoSuscriptor → Surtidor:** `RespuestaCobro(ok)`

Caso 3: Uso en nueva estación (ya conocida por otras)

1. **Surtidor → NodoSuscriptor:** `Cobrar(tarjeta, monto)`

2. **NodoSuscriptor** no conoce la tarjeta → propaga a **vecinos**

3. **NodoVecino** con registro → responde con `Registro(tarjeta)`

4. **NodoSuscriptor** almacena el registro recibido

5. **NodoSuscriptor → NodoLíder:** `Suscripción`

6. **NodoSuscriptor** valida límite de tarjeta local:
   Si **excede** → `RespuestaCobro(false)`
   Si **OK** → **NodoSuscriptor → NodoLíder:** `Cobrar(tarjeta, monto)`

7. **NodoLíder → NodoCuenta:** `Cobrar(tarjeta, monto)`

8. **NodoCuenta** valida límite global:
   Si **excede** → `RespuestaCobro(false)`
   Si **OK** → `RespuestaCobro(true)`

9. **NodoLíder** envía `Actualización` a todos los suscriptores *(incluido el nuevo)*

10. **NodoLíder → NodoSuscriptor:** `RespuestaCobro(ok)`

11. **NodoSuscriptor** actualiza su `TTL` y registro

12. **NodoSuscriptor → Surtidor:** `RespuestaCobro(ok)`

Protocolo de comunicación

Por tratarse de un sistema distribuido, los nodos obviamente no comparten memoria, si no que se comunican 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. Para los mensajes descriptos en la descripción del modelo de autores, se propone la siguiente estructura:

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

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

- **ID de un nodo.** Sabemos que hay 1600 nodos por lo que bastarían 11 bits para representarlos a todos. Para no hacer operaciones bit-wise y para tener margen para muchas más estaciones usamos 2 bytes. Los IDs podrían ser caracteres ascii o números enteros, es indistinto.
- **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).
- **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)** tanto para la comunicación local entre *surtidores*, como para la comunicación entre los distintos nodos

distribuidos del sistema (*suscriptores, líderes y cuentas*).
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 esencial para que los nodos se enteren si sus pares fallan y actuen en concecuencia.

Comunicación local

Dentro de cada estación, los surtidores se conectan al nodo central mediante TCP sobre la red local (LAN).
Este canal asegura que los mensajes `Cobrar` y las respuestas de autorización se transmitan sin pérdidas ni duplicaciones, manteniendo la coherencia del registro de ventas.

Comunicación entre nodos

Las estaciones y los distintos nodos del sistema intercambian información mediante TCP, manteniendo sincronizados los registros de tarjetas y cuentas.
El uso de TCP facilita la detección de desconexiones, el control de flujo y la confirmación explícita de entrega, reduciendo la complejidad de los mecanismos de replicación y actualización distribuidos.