

# Trabajo Práctico

## Introducción

YPF Ruta es un sistema que permite a las empresas centralizar el pago y el control de gasto de combustible 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 distribuidas 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.

## Objetivo

La empresa YPF nos contrata para implementar un conjunto de aplicaciones que modelen el sistema descripto. Se decidió que dicha implementación se realice íntegramente usando Rust por su seguridad y eficiencia en el manejo de memoria.

## Requerimientos

- Cada uno de los surtidores donde los vehículos cargan combustible perteneciente a una estación puede registrar una venta contra una tarjeta Ruta.
- Los administradores de empresa pueden visualizar en todo momento los gastos registrados por sus vehículos en tiempo real y el estado de cuenta total. Además pueden establecer límites para la cuenta de la empresa y para vehículos en particular.
- Las estaciones, y por lo tanto todos sus surtidores, se encuentran repartidas por todo el país, en algunos casos con conectividad limitada. En el caso que una estación se encuentre sin conexión, se deberán de tener que poder registrar ventas con Ruta igualmente, las cuales se facturarán luego cuando la conexión se recupere.
- El sistema debe intentar minimizar la cantidad de mensajes que viajan por toda la red; por ejemplo enviando mensajes sólo a los nodos que se encuentren cercanos por ubicación.



# 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.
- Por lo menos alguna, si no todas las aplicaciones implementadas deben funcionar utilizando el **modelo de actores**.
- Cada instancia de cada aplicación debe ejecutarse en un proceso independiente.
- En el modelado de la solución se deberán utilizar **al menos dos** de las herramientas de concurrencia distribuida mostradas en la cátedra. Por ejemplo: exclusiones mutuas distribuidas, elección de líder, algoritmos de anillo, commits de dos fases, etc.
- No se permite utilizar **crates** externos, salvo los explícitamente mencionados en este enunciado, los utilizados en las clases, o los autorizados expresamente por los profesores.
- 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.

## Entregas

La resolución del presente proyecto es en grupos de cuatro integrantes. No se aceptan grupos de otra cantidad de miembros.

Las entregas del proyecto se realizarán mediante Github Classroom. Cada grupo tendrá un repositorio disponible para hacer diferentes commits con el objetivo de resolver el problema propuesto.

### Primera entrega: Diseño

Deberán entregar un informe en formato Markdown en el `README.md` del repositorio que contenga una explicación del diseño y de las decisiones tomadas para la implementación de la



solución, así como diagramas de threads y procesos, y la comunicación entre los mismos; y diagramas de las entidades principales.

De cada entidad se debe describir:

- Finalidad general
- Su estado interno, como uno o varios structs en pseudocódigo de Rust.
- Mensajes que recibe, struct del payload de los mismos y como reaccionará cuando los recibe.
- Mensajes que envía, struct del payload de los mismos y hacia quienes son enviados.
- Protocolos de transporte (TCP/UDP) y de aplicación que utiliza para comunicarse.
- Casos de interés (felices, caídas)

Se recomienda fuertemente que las implementaciones de las ideas de diseño se encuentren realizadas mínimamente, para validar el mismo.

### **Fecha máxima de Entrega: 5 de Noviembre de 2025**

Presentar como un pull-request del README.md.

El diseño será evaluado por la cátedra y de no presentarse a término el trabajo quedará automáticamente desaprobado.

La cátedra podrá solicitar correcciones que deberán realizarse en el mismo diseño, y puntos de mejora que deberán tenerse en cuenta durante la implementación.

Se podrán hacer commits hasta el día de la entrega a las 19 hs Arg, luego el sistema automáticamente quitará el acceso de escritura.

### **Segunda entrega: Código final - 25 de Noviembre de 2025**

- Deberá incluirse el código de la solución completa.
- Deberá actualizarse el readme, con una sección dedicada a cambios que se hayan realizado desde la primera entrega. Además debe incluir cualquier explicación y/o set de comandos necesarios para la ejecución de los programas.

### **Presentación final: 25 y 26 de Noviembre, 2 y 3 de Diciembre de 2025.**

Cada grupo presentará presencialmente a un profesor de la cátedra su trabajo. La fecha y horario para cada grupo serán definidos oportunamente.

La presentación deberá incluir un resumen del diseño de la solución y la muestra en vivo de las aplicaciones funcionando, demostrando casos de interés.



Todos los integrantes del grupo deberán exponer algo. La evaluación es individual.

La cátedra podrá solicitar luego correcciones donde se encuentren errores severos, sobre todo en el uso de herramientas de concurrencia; o bien desviaciones respecto al diseño pactado inicialmente.

De tener correcciones para realizar, las mismas deben realizarse y aprobarse con anterioridad a la presentación a examen final.

## **Evaluación**

### **Principios teóricos y corrección de bugs**

Los estudiantes presentarán el diseño, código y comportamiento en vivo de su solución, con foco en el uso de las diferentes herramientas de concurrencia.

Deberán poder explicar, desde los conceptos teóricos vistos en clase, cómo se comportará potencialmente su solución ante problemas de concurrencia (por ejemplo ausencia de deadlocks).

En caso de que la solución no se comportara de forma esperada, deberán poder explicar las causas y sus posibles rectificaciones.

### **Casos de prueba**

Se someterá a la aplicación a diferentes casos de prueba que validen la correcta aplicación de las herramientas de concurrencia y la resiliencia de las distintas entidades.

### **Informe**

El README de su repositorio funcionará como informe del trabajo y deberá contener toda la información requerida tanto para la primera como para la segunda entrega.

### **Organización del código**

El código debe organizarse respetando los criterios de buen diseño y en particular aprovechando las herramientas recomendadas por Rust. Se prohíbe el uso de bloques



`unsafe .`

## Tests automatizados

La entrega debe contar con tests automatizados que prueben diferentes casos. Se considerará en especial aquellos que pongan a prueba el uso de las herramientas de concurrencia.

## Presentación en término

El trabajo deberá entregarse para la fecha estipulada. La presentación fuera de término sin coordinación con antelación con el profesor influirá negativamente en la nota final.