# Trabajo Práctico

## Introducción

ConcuRide es una nueva aplicación para conectar conductores y pasajeros. Gracias a su inovadora implementación distribuida, permitirá reducir los costos y apuntar a ser líder en el mercado.

Los pasajeros tendrán una app donde podrán elegir el destino, y los choferes una app donde recibirán y podrán aceptar o denegar viajes.

## Objetivo

Deberán implementar un conjunto de aplicaciones en Rust que modele el sistema.

## Requerimientos

- Una aplicación modelará la app para los pasajeros. Por simplificación, las coordenadas de origen y destino del viaje se modelarán con un par de enteros pequeños. Por ejemplo un viaje puede ser desde (3, 5) hacia (24,16). La app informará cuando un conductor acepte el viaje, y luego cuando esté terminado. Habrán varias instancias que se ejecutarán concurrentemente, una por cada pasajero.
- Otra aplicación simulará a los conductores. Los mismos irán informando la ubicación actual del vehículo (simulada aleatoreamente o bien con un control, lo que simplifique los casos de prueba). El conductor recibirá ofertas de viajes cercanos a su posición actual (por ejemplo con una distancia de 5 entre el origen y su posición). Un único conductor puede recibir la oferta del viaje por cada vez; y tiene un tiempo corto para aceptarla. La aceptación o rechazo se modelará con un random. Al aceptar, el tiempo viaje se modelará con un sleep, luego del cual se terminará el viaje y el conductor quedará disponible para tomar otro.
- Se debe considerar la autorización del pago al momento de solicitar el viaje, y el cobro efectivo al momento de llegar a destino. Modelar el gateway de pagos con una aplicación

simple que loguea. Considerar que al momento de autorizar el pago, el gateway puede rechazar la tarjeta aleatoriamente con una probabilidad.

- El sistema debe ser resiliente, soportando la caida de algunas instancias de las aplicaciones.
- El sistema debe intentar minimizar la cantidad de mensajes que viajan por toda la red; por ejemplo enviando mensajes solo 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 una o mas de las herramientas de concurrencia distribuida mostradas en la cátedra. Por ejemplo exclusiones mutuas distribuidas, elección de líder, algorítmos 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 tres integrantes.

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, asi 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 pseudocodigo 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.
- Protolos 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: 13 de Noviembre de 2024**

Presentar como un pull-request del README.md.

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

La cátedra podrá solicitar correciones 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 - 3 de Diciembre de 2024

- 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: 3, 4, 10 y 11 de Diciembre

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 interes.

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

## Evaluación

### Principios teóricos y corrección de bugs

Los alumnos 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 informe debe poder dar cuenta de todas las decisiones tomadas para implementar la solución, incluyendo diferentes diagramas y se debe poder utilizar como soporte para que el equipo presente su trabajo.

## Organización del código

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