Práctica 4 – Sistema Distribuido Uber

Introducción

En esta práctica se implementó una simulación de un sistema de transporte similar a Uber, utilizando gRPC como protocolo de comunicación entre un servidor Python y uno o más clientes Flutter. Se manejaron conceptos de concurrencia, serialización con Protocol Buffers y despliegue multi-emulador para pruebas paralelas.

Video de Evidencia

Enlace al video de evidencia: https://youtu.be/rtb_X-o-HsU

Tecnologías utilizadas

- Python 3

- Flutter

- gRPC

- Protocol Buffers

- Emuladores Android

Arquitectura del sistema

El sistema se basa en una arquitectura cliente-servidor. El cliente solicita viajes y termina viajes, mientras que el servidor se encarga de asignar autos, calcular distancias, precios y mantener el estado global de los vehículos. La comunicación se realiza mediante gRPC utilizando protobuf para definir los mensajes.

Implementación del servidor

El servidor fue escrito en Python utilizando grpcio. Las funciones principales son:

- solicitarViaje: Asigna el auto más cercano disponible.

- terminarViaje: Calcula el precio con base en la distancia recorrida.

- estadoServicio: Devuelve el estado actual de todos los autos.

Implementación del cliente

El cliente Flutter contiene una interfaz simple con tres botones:

- Estado del servicio

- Solicitar viaje

- Terminar viaje

Además, incluye validaciones para evitar pérdida de autos y mantiene el ID del auto en viaje.

Concurrencia y múltiples emuladores

Se probaron múltiples instancias de emuladores para validar que los viajes se reflejan correctamente en todos los clientes. El servidor mantiene un estado global compartido correctamente sincronizado.

Errores comunes y soluciones

- Error al compilar Flutter por archivos en uso: se solucionó cerrando procesos con `taskkill` o reiniciando.

- Flutter `build` bloqueado: se liberó la carpeta usando PowerShell forzando cierre de procesos.

- Estado inconsistente del servidor: se corrigió usando `inicio_viaje` para trackear posición inicial.

Definición del archivo .proto

El archivo `.proto` define los mensajes y servicios utilizados entre el servidor y los clientes:

```
syntax = "proto3";

package uber;

import "google/protobuf/empty.proto";

service UberService {
  rpc solicitarViaje(Posicion) returns (InfoAuto);
  rpc terminarViaje(TerminaViajeArgs) returns (ResultadoViaje);
  rpc estadoServicio(google.protobuf.Empty) returns (InfoServicio);
}
```

```
message Posicion {
  double latitud = 1;
  double longitud = 2;
}

message InfoAuto {
  int32 id = 1;
  Posicion posicion = 2;
  double distancia = 3;
  double precio = 4;
}

message TerminaViajeArgs {
  int32 id_auto = 1;
  Posicion posicion = 2;
}

message InfoServicio {
  int32 total_viajes = 1;
  repeated Auto autos = 2;
}

message Auto {
  int32 id = 1;
  Posicion posicion = 2;
  bool disponible = 3;
}

message ResultadoViaje {
  double precio = 1;
}
```

Servidor Python (gRPC)

El servidor implementa la lógica de asignación de viajes, terminación y consulta del estado. Utiliza `threading.Lock()` para evitar condiciones de carrera al modificar recursos compartidos entre clientes concurrentes:

Fragmentos relevantes del servidor:

```
self.lock = threading.Lock()
```

```
with self.lock:
  for auto in self.autos:
    if auto.disponible:
      distancia = calcular_distancia(auto.posicion, cliente_pos)
      ...
```

El servidor almacena la posición inicial del viaje para poder calcular correctamente el precio final al terminar el viaje. Además, reduce la lista de autos disponibles para reflejar el uso actual del sistema en tiempo real.

Cliente Flutter

El cliente Flutter consume los servicios gRPC y permite al usuario: solicitar un viaje, terminarlo y consultar el estado del sistema. El cliente evita iniciar múltiples viajes sin finalizar uno primero.

Se utiliza una validación en `solicitarViaje()` para impedir que un auto vuelva a ser asignado si aún no termina su viaje:

```
if (ultimoAutoAsignado != null) {
 setState(() {
  estado = "Primero termina el viaje actual.";
 });
 return;
}
```

Conclusión

Esta práctica integró múltiples conceptos clave de los sistemas distribuidos: comunicación remota con gRPC, manejo de concurrencia con hilos, sincronización de estado compartido, y pruebas simultáneas en múltiples instancias. Esta práctica ayudó a consolidar conocimientos sobre sistemas distribuidos, comunicación con gRPC y sincronización de estados entre múltiples clientes.