

SD - Segunda Versión

Easy Cab



2024/2025

Índice

Introducción.....	3
Informe de desarrollo.....	3
Central.....	3
Clases principales.....	3
Funcionalidades principales.....	4
Digital Engine.....	5
Clase principal.....	5
Funcionalidades principales.....	5
Sensor.....	6
Clase principal.....	6
Funcionalidades principales.....	6
Cliente.....	6
Clase principal.....	6
Funcionalidades principales.....	7
City Traffic Control.....	7
Clase principal.....	7
Funcionalidades principales.....	7
Registry.....	8
Clase principal.....	8
Funcionalidades principales.....	8
Guía de despliegue.....	8
Linux.....	8
Windows.....	9
Resultados.....	11
Funcionamiento básico de la aplicación.....	11
Comandos de la central.....	14
Incidencias.....	16
Resiliencia.....	17
Registro.....	18
Consulta de tráfico.....	19
Auditoría.....	19

Introducción

En esta práctica se nos pide en nuestra aplicación distribuida que simula la gestión de una flota de taxis en un mapa o ciudad ficticia, agregar un módulo que informe del estado del tráfico, implementar la autenticación entre la central y los taxis, así como un cifrado en este canal y un registro de auditoría de los eventos que sucedan. El objetivo de esta práctica es que aprendamos sobre el consumo y la creación de APIs, la seguridad en sistemas distribuidos, uso de estructuras concurrentes y bloqueos, autenticación y uso de claves simétricas, y a proporcionar interfaces HTML sencillas entre otros.

Los componentes de nuestro sistema Easy Cab son:

- Central (EC_Central.py): Componente principal que implementa la lógica central y maneja todo el sistema.
- Digital Engine (EC_DE.py): Componente que maneja la lógica de los taxis y también la comunicación con el control central.
- Sensores del taxi (EC_S.py): Componente que simula los sensores de los taxis autónomos y que envía información sobre el estado de estos a los mismos mediante sockets.
- Clientes (Customer.py): Componente utilizado por los clientes para solicitar los servicios de un taxi.
- City Traffic Control (EC_CTC.py): Componente utilizado para informar a la central sobre el estado del tráfico en una ciudad.
- Registry (EC_Registry.py): Componente que se encarga del registro de los taxis en el sistema.
- API Central (API_Central.py): Componente fundamental para el correcto funcionamiento del front end.

Informe de desarrollo

Central

Este componente es el desarrollado en EC_Central.py que maneja la lógica principal de la aplicación. Gestiona la comunicación entre los taxis y los clientes y proporciona una interfaz gráfica para la monitorización en tiempo real.

Clases principales

- Clase Central. Sus características principales son:
 - Gestión de la comunicación mediante kafka.
 - Mantenimiento del estado del sistema.

- Coordinación de servicios de taxi y notificación a clientes sobre el estado de sus solicitudes. Comunicación segura con taxis mediante cifrado AES.
 - Actualización del mapa y la tabla en tiempo real.
 - Gestión de la base de datos SQLite.
 - Manejo de mensajes perdidos durante la desconexión.
 - Monitorización de clientes. Detecta clientes desconectados y los elimina del sistema automáticamente.
 - Gestión del tráfico con consulta periódica desde el módulo EC_CTC.
 - Cifrado y autenticación.
 - Auditoría que registra diversos sucesos en la aplicación.
- Clase StatusWindow. Sus características principales son:
 - Visualización del mapa.
 - Tablas de estado para taxis y clientes.
 - Actualización en tiempo real de la información.
- Clase PygameWidget. Widget cuyas características son:
 - Renderizar el mapa del sistema.
 - Integrar Pygame con PyQt5.
 - Proporcionar visualización de las posiciones en tiempo real.

Funcionalidades principales

- Gestión de servicios: Implementa un sistema de asignación de taxis a los clientes. Además mantiene un registro de cada servicio en la base de datos y gestiona el proceso de recogida y entrega de cada uno.
- Sistema de comunicaciones: Implementa comunicación entre los componentes por medio de Kafka Producer/Consumer. También dispone de unos topics principales que son:
 - taxi_requests: para solicitudes de servicio del cliente a la central.
 - taxi_status: para actualizaciones del estado de los taxis enviadas a la central.
 - taxi_instructions: para los comandos que envía la central al taxi.
 - map_updates: para actualizaciones del mapa que envía la central al taxi.
 - customer_responses: para respuestas a clientes.
- Gestión de estados: Mantiene el estado actual de las posiciones de los taxis, servicios, ubicación de clientes y asignaciones taxi a pasajero.
- Visualización del mapa: Representa gráficamente el mapa, con las ubicaciones, los taxis y los clientes en tiempo real. Además de una tabla de estados de taxis y otra de clientes.
- Seguridad y cifrado: Cifrado AES para comunicación segura y validación de tokens únicos generados para cada taxi. Cifrado y descifrado de mensajes con claves simétricas.
- Gestión del tráfico: Estado del tráfico recibido desde EC_CTC, con asignaciones automáticas en caso de interrupciones (KO). Retorno de taxis a la base y uso de

estados previos si no hay conexión con EC_CTC. Cambio dinámico de ciudad a través de solicitudes POST.

- Auditoría: Registra con fecha y hora, eventos que abarcan desde autenticación, incidencias durante el servicio, cambios de la situación del tráfico y errores de diferente tipo.
- Base de datos: utiliza SQLite con dos tablas principales:
 - Tabla de taxis.
 - Tabla de servicios.
- Manejo de excepciones y cierre controlado de la aplicación. Notifica al taxi y al cliente cuando el cierre se lleva a cabo.

Digital Engine

Digital Engine (EC_DE.py) es un componente que representa al comportamiento de cada taxi. Es responsable de gestionar el estado, la posición y el comportamiento de cada uno de los taxis dentro de la aplicación. Mantiene la comunicación con la central y los sensores.

Clase principal

La clase Digital Engine gestiona todas las funcionalidades principales de los taxis. Sus características son:

- Conexiones kafka para la comunicación distribuida.
- Manejo de sensores mediante sockets TCP/IP. Implementa un servidor TCP/IP para recibir y procesar los mensajes de estado enviados por los sensores de cada taxi.
- Posición y navegación de los taxis.
- Interfaz gráfica que muestra un mapa enviado por la central. Utiliza pygame para renderizar el mapa que indica la posición de taxis y clientes, y el estado de los taxis.
- Sistema de estados para el control del taxi.
- Permite establecer un destino principal y un siguiente destino para cada taxi.
- Posibilidad de registrarse y darse de baja.

Funcionalidades principales

- Comunicación con kafka Producer/Consumer. Recepción de instrucciones de la central a través del topic taxi_instructions. Recepción de actualizaciones del mapa a través del topic map_updates.
- Comunicación con sensores mediante servidor TCP/IP con gestión automática de reconexiones y protocolo de mensajes con formato <STX>estado<ETX>.
- Gestión de estado, que puede ser OK, KO o RUN.
- Sistema de navegación que consiste en un algoritmo de movimiento paso a paso hacia el destino, con actualización continua de la posición y soporte para varios destinos.
- Visualización gráfica del mapa que muestra el mapa, la posición y el estado de los taxis mediante colores, los clientes y las ubicaciones. Utiliza Pygame para renderizar el mapa y mostrar la información relevante.

- Procesamiento de instrucciones recibidas desde la central. Maneja diferentes tipos de comandos como mover, detener, reanudar o cambiar el destino de cada taxi.
- Registro: Lee la URL del Registry desde un archivo .txt y envía una solicitud POST para registrar el taxi. El taxi puede darse de baja. Utiliza JSON para enviar su id y realizar estos trámites.
- Las comunicaciones con EC_Registry utilizan HTTPS validando el certificado del servidor.
- Utiliza los datos de EC_CTC que le envía la central. Puede volver a base o pausar el servicio.

Sensor

El componente Sensor (EC_S) simula los sensores que envían información a los taxis y reportan el estado operativo de estos. Los sensores mantienen una comunicación constante con Digital Engine enviando actualizaciones sobre el estado e incidencias.

Clase principal

La clase Sensors gestiona todas las funcionalidades de este componente. Sus características incluyen:

- Gestión de las conexiones TCP/IP con Digital Engine.
- Sistema de monitorización continua de los taxis.
- Simulación de incidencias mediante entradas de teclado.
- Sistema de reconexión automática

Funcionalidades principales

- Gestión de conexiones TCP/IP persistente mediante sockets con Digital Engine.
- Envío cada segundo de mensajes de estados, indicando si el funcionamiento es normal (OK) o hay alguna incidencia (KO).
- Simulación de incidencias: tecla 'i', activa incidencia (estado KO), y tecla 'q' desactiva la incidencia (estado OK). Los cambios de estado los obtenemos de inmediato por consola.

Cliente

El componente Cliente (EC_Customer.py) simula el comportamiento de un cliente en nuestra aplicación de gestión de taxis. Este componente permite a los clientes solicitar un servicio de taxi, en el cual pueden especificar ubicaciones de recogida y destino.

Clase principal

La clase Customer gestiona todas las funcionalidades de este componente: Sus características son:

- Gestión de las conexiones Kafka para el envío y la recepción de mensajes.

- Sistema de procesamiento de solicitudes desde un archivo JSON.
- Seguimiento de la posición actual del cliente.
- Sistema de reintento automático para solicitudes sin respuesta.
- Control del estado de las solicitudes y respuestas.
- Detección automática de fallos del servidor central y reconexión periódica.
- Recuperación de solicitudes pendientes tras conexión.

Funcionalidades principales

- Comunicación mediante kafka: Envío de solicitudes de taxi a través del topic 'taxi requests', y recepción de respuestas mediante el topic 'customer_responses'.
- Gestión de solicitudes: actualización automática de la posición del cliente después de un viaje completado. Las solicitudes se leen desde un Archivo JSON de configuración, y se envían según el orden establecido.
- En el sistema de reintentos, si no hay respuesta en 4 segundos se reenvía la solicitud, y se cancela el reintento al recibir una respuesta válida.
- En el procesamiento de respuestas, manejamos asignaciones exitosas de taxis, además se gestionan los casos de no disponibilidad y la confirmación de llegada al destino.

City Traffic Control

Es el componente (EC_CTC) que determina si es viable circular por una ciudad en la que se está prestando un servicio. Implementa un control del tráfico urbano que consulta datos meteorológicos en tiempo real y proporciona un estado del tráfico basado en la temperatura.

Clase principal

La clase EC_CTC controla toda la lógica principal del control de tráfico y sus características principales son:

- Lectura de API Key.
- Obtención de datos meteorológicos.
- Manejo de errores.
- Gestión de ciudad predeterminada.

Funcionalidades principales

- Puede leer una clave API desde un archivo apikey.txt para interactuar con servicios externos, como OpenWeather. Obtiene datos de esta API para obtener el clima de una ciudad específica y retorna la temperatura actual.
- Utiliza Flask para ofrecer una API web que permite interactuar con el sistema de forma remota.
- Maneja errores comunes, como claves no válidas o archivos inexistentes. También identifica errores relacionados con la API como problemas de red.

- Mantiene por defecto la ciudad de Torrevieja desde donde se consultan los datos meteorológicos. Permite cambiar esta ciudad de manera segura para evitar conflictos utilizando threading.Lock.

Registry

El componente Registry (EC_Registry) se encarga de gestionar el registro y la eliminación de taxis de nuestra aplicación. Proporciona una interfaz web para realizar estas operaciones y expone una API para interactuar con la base de datos central del sistema de taxis.

Clase principal

Algunas de las características de la clase principal EC_Registry son las siguientes:

- Gestión de la base de datos.
- Seguridad con SSL.
- Interfaz web amigable:
- API RESTful.
- Conexión con la base de datos.
- Manejo de errores.

Funcionalidades principales

- Registro de taxis: Agrega un taxi a la base de datos con coordenada (0,0), devolviendo un mensaje de éxito o de error dependiendo de si el taxi está registrado o no.
- Eliminación de taxis: Busca un taxi en la base de datos y lo elimina si existe, notificando al usuario en caso de éxito o si el taxi no se encuentra.
- Interfaz web con formularios: Un formulario para registrar taxis y otro para darlos de baja.
- API REST: Permite realizar las mismas operaciones a través de solicitudes JSON, facilitando la integración con otros sistemas o aplicaciones.
- Conexión segura: Se garantiza que las comunicaciones entre el servidor y los clientes estén cifradas gracias al protocolo HTTPS.
- Manejo de errores: Verifica la ausencia del id del taxi y detecta y gestiona conflictos como intentar registrar un taxi ya existente.

Guía de despliegue

Hemos desarrollado tanto una guía de despliegue para linux como para windows:

Linux

- Primero, desplegamos los servidores:

- 1 ./bin/zookeeper-server-start.sh ./config/zookeeper.properties
- 2 ./bin/kafka-server-start.sh ./config/server.properties

- Segundo, el productor y el consumidor, creamos sus topics:

```
-1 ./bin/kafka-topics.sh --create --topic customer_responses --bootstrap-server localhost:9092
```

```
-2 ./bin/kafka-topics.sh --create --topic taxi_requests --bootstrap-server localhost:9092
```

```
-3 ./bin/kafka-topics.sh --create --topic taxi_status --bootstrap-server localhost:9092
```

```
-4 ./bin/kafka-topics.sh --create --topic taxi_instructions --bootstrap-server localhost:9092
```

```
-5 ./bin/kafka-topics.sh --create --topic map_updates --bootstrap-server localhost:9092
```

- Tercero, desplegamos el programa.

```
-1 python3 EC_Central.py --listen_port 8000 --kafka_bootstrap_servers localhost:9092 --locations_file EC_locations.json
```

```
-2 python3 EC_DE.py localhost 8000 localhost:9092 9000
```

```
-3 python3 EC_Costumer.py localhost:9092 a EC_Requests.json 10 10
```

```
-4 python3 EC_S.py localhost 9000
```

```
-5 python3 EC_CTC.py
```

```
-6 python3 EC_Registry.py
```

```
-7 python3 API_Central.py
```

COMANDO PARA VER LOS TOPICS CREADOS:

```
bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

COMANDO PARA VER LOS MENSAJES DE UNO DE LOS TOPICS:

```
bin/kafka-console-consumer.sh --topic <nombre_del_topic> --from-beginning --bootstrap-server localhost:9092
```

COMANDO PARA ENTRAR EN EL ENTORNO VIRTUAL:

```
source myenv/bin/activate
```

ESQUEMA DE LOS TOPICS

CLIENTE -> taxi_requests -> CENTRAL

CLIENTE <- customer_responses <- CENTRAL

TAXI -> taxi_status -> CENTRAL

TAXI <- taxi_instructions <- CENTRAL

TAXI <- map_updates <- CENTRAL

Windows

- Primero, desplegamos los servidores:

```
-1 .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

-2 .\bin\windows\kafka-server-start.bat .\config\server.properties

- Segundo, el productor y el consumidor, creamos sus topics:

-1 .\bin\windows\kafka-topics.bat --create --topic customer_responses
--bootstrap-server localhost:9092

-2 .\bin\windows\kafka-topics.bat --create --topic taxi_requests --bootstrap-server
localhost:9092

-3 .\bin\windows\kafka-topics.bat --create --topic taxi_status --bootstrap-server
localhost:9092

-4 .\bin\windows\kafka-topics.bat --create --topic taxi_instructions --bootstrap-server
localhost:9092

-5 .\bin\windows\kafka-topics.bat --create --topic map_updates --bootstrap-server
localhost:9092

- Tercero, desplegamos el programa.

-1 python EC_Central.py --listen_port 8000 --kafka_bootstrap_servers localhost:9092
--locations_file EC_locations.json
-2 python EC_DE.py localhost 8000 localhost:9092 9000
-3 python EC_Costumer.py localhost:9092 a EC_Requests.json 10 10
-4 python EC_S.py localhost 9000
-5 python EC_Registry.py
-6 python EC_CTC.py
-7 python API_Central.py

COMANDO PARA VER LOS TOPICS CREADOS:

.\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092

COMANDO PARA VER LOS MENSAJES DE UNO DE LOS TOPICS:

.\bin\windows\kafka-console-consumer.bat --topic <nombre_del_topic> --from-beginning
--bootstrap-server localhost:9092

COMANDO PARA ENTRAR EN EL ENTORNO VIRTUAL:

myenv\Scripts\activate

ESQUEMA DE LOS TOPICS

CLIENTE -> taxi_requests -> CENTRAL

CLIENTE <- customer_responses <- CENTRAL

TAXI -> taxi_status -> CENTRAL

TAXI <- taxi_instructions <- CENTRAL

TAXI <- map_updates <- CENTRAL

Resultados

Las primeras pruebas realizadas son las que realizamos para la anterior entrega que las he dejado pues el funcionamiento es el mismo. Después realizaremos distintas pruebas mostrando el funcionamiento de algunas de las implementaciones realizadas para esta entrega.

Funcionamiento básico de la aplicación

Primeramente debemos iniciar zookeeper y kafka, con los comandos mencionados en la guía de despliegue. Cada uno en una terminal distinta:

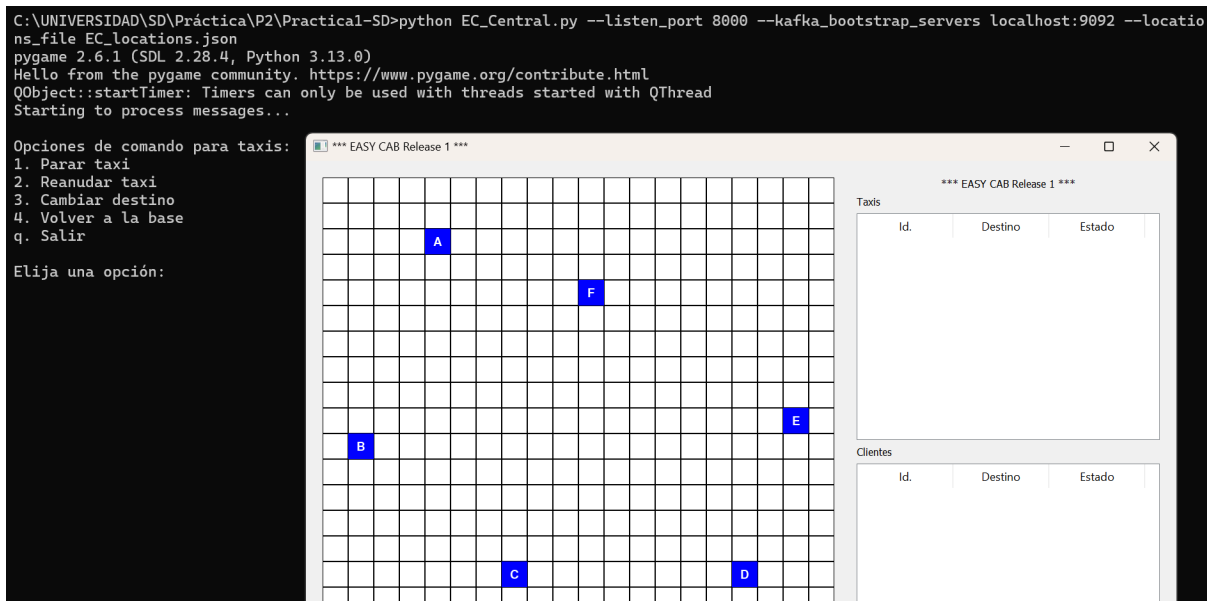
```
[2024-10-31 13:25:39,818] INFO Snapshot loaded in 39 ms, highest zxid is 0x283, digest is 349806167597 (org.apache.zookeeper.server.ZKDatabase)
[2024-10-31 13:25:39,819] INFO Snapshotting: 0x283 to \tmp\zookeeper\version-2\snapshot.283 (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
[2024-10-31 13:25:39,823] INFO Snapshot taken in 5 ms (org.apache.zookeeper.server.ZooKeeperServer)
[2024-10-31 13:25:39,833] INFO zookeeper.request_throttler.shutdownTimeout = 10000 ms (org.apache.zookeeper.server.RequestThrottler)
[2024-10-31 13:25:39,833] INFO PrepRequestProcessor (sid:0) started, reconfigEnabled=false (org.apache.zookeeper.server.PrepRequestProcessor)
[2024-10-31 13:25:39,846] INFO Using checkIntervalMs=60000 maxPerMinute=10000 maxNeverUsedIntervalMs=0 (org.apache.zookeeper.server.ContainerManager)
[2024-10-31 13:25:59,291] INFO Expiring session 0x10068b9124a0001, timeout of 18000ms exceeded (org.apache.zookeeper.server.ZooKeeperServer)
[2024-10-31 13:25:59,298] INFO Creating new log file: log.284 (org.apache.zookeeper.server.persistence.FileTxnLog)
```

```
group.GroupMetadataManager)
[2024-10-31 13:26:08,147] INFO [GroupMetadataManager brokerId=0] Finished loading offsets and group metadata from __consumer_offsets-43 in 80 milliseconds for epoch 0, of which 80 milliseconds was spent in the scheduler. (kafka.coordinator.group.GroupMetadataManager)
[2024-10-31 13:26:08,147] INFO [GroupMetadataManager brokerId=0] Finished loading offsets and group metadata from __consumer_offsets-13 in 79 milliseconds for epoch 0, of which 79 milliseconds was spent in the scheduler. (kafka.coordinator.group.GroupMetadataManager)
[2024-10-31 13:26:08,148] INFO [GroupMetadataManager brokerId=0] Finished loading offsets and group metadata from __consumer_offsets-28 in 80 milliseconds for epoch 0, of which 79 milliseconds was spent in the scheduler. (kafka.coordinator.group.GroupMetadataManager)
```

Ahora que tenemos ambos corriendo, el siguiente paso sería crear los topics, tal como se indica en la guía de despliegue. Como yo ya los tengo creados, mostraré cuales son:

```
C:\UNIVERSIDAD\SD\Práctica\P2\Practical-SD>kafka\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092
SD
__consumer_offsets
customer_responses
map_updates
taxi_instructions
taxi_requests
taxi_status
```

A continuación desplegamos la aplicación:



Podemos observar el mapa con las localizaciones, pero no se observa ningún taxi ni ningún cliente porque no los he ejecutado aún.

Ejecutamos EC_DE.py:

```
C:\UNIVERSIDAD\SD\Práctica\P2\Practical-SD>python EC_DE.py localhost 8000 localhost:9092 9000 1
pygame 2.6.1 (SDL 2.28.4, Python 3.13.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
Initializing Digital Engine for taxi 1...
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'KO', 'position': [1, 1]}
Connected to Central via Kafka
Subscribed to topics: {'taxi_instructions'}
PyGame initialized successfully
Starting to process map updates...
Listening for sensor connections on port 9000
Starting to process instructions...
Todos los threads iniciados correctamente
Map updated - Size: 20x20
Taxis: 0, Customers: 0
Map updated - Size: 20x20
Taxis: 1, Customers: 0
```

El taxi aparece en la posición inicial (1,1), con status (KO) porque no se está moviendo aún. También se dibuja un mapa que es enviado de la central al taxi y es exactamente idéntico en todo momento ya que se envía cada vez que se actualiza y hay cambios.

Podemos observar en la tabla de la central como se ha creado una nueva entrada para este taxi.

*** EASY CAB Release 1 ***			
Taxis			
	Id.	Destino	Estado
1	1	A	KO

El siguiente paso es ejecutar EC_Customer.py:

```
C:\UNIVERSIDAD\SD\Práctica\P2\Practical1-SD>python EC_Customer.py localhost:9092 a EC_Requests.json 10 10
Customer a starting at position [10, 10]...
DEBUG: Sending request: {'type': 'pedir_taxi', 'customer_id': 'a', 'pickup': [10, 10], 'destination': 'A', 'timestamp': 1730378848.7284198}
Sent request for pickup at [10, 10] and destination A
DEBUG: Waiting for response...
DEBUG: No response in 4 seconds, retrying request...
DEBUG: Sending request: {'type': 'pedir_taxi', 'customer_id': 'a', 'pickup': [10, 10], 'destination': 'A', 'timestamp': 1730378852.72998}
Sent request for pickup at [10, 10] and destination A
DEBUG: Received response: {'type': 'no_taxi_available', 'customer_id': 'a', 'pickup': [10, 10], 'destination': 'A'}
Request denied: No taxis available
```

Como aún no hemos desplegado los sensores, el Cliente pide un taxi pero no hay taxis disponibles. Desplegamos los sensores:

```
C:\UNIVERSIDAD\SD\Práctica\P2\Practical1-SD>python EC_S.py localhost 9000
Connected to Digital Engine at localhost:9000
Sent: <STX>OK<ETX>
Sent: <STX>OK<ETX>
Sent: <STX>OK<ETX>
Sent: <STX>OK<ETX>
Sent: <STX>OK<ETX>
```

Los sensores indican que el funcionamiento del taxi es correcto (OK) periódicamente y el resto de la aplicación comienza a moverse hacia el cliente para recogerlo y llevarlo al destino. Si hay más solicitudes las procesa y cuando termina de procesarlas todas termina la ejecución.

```
Request accepted: Taxi 1 assigned for pickup at [10, 10] and destination A
Arrived at destination with taxi 1
Updated customer position to: [4, 2]
Sent request for pickup at [4, 2] and destination C
Request accepted: Taxi 1 assigned for pickup at [4, 2] and destination C
Arrived at destination with taxi 1
Updated customer position to: [7, 15]
Sent request for pickup at [7, 15] and destination F
Request accepted: Taxi 1 assigned for pickup at [7, 15] and destination F
Arrived at destination with taxi 1
Updated customer position to: [10, 4]
Sent request for pickup at [10, 4] and destination D
Request accepted: Taxi 1 assigned for pickup at [10, 4] and destination D
Arrived at destination with taxi 1
Updated customer position to: [16, 15]
Sent request for pickup at [16, 15] and destination E
Request accepted: Taxi 1 assigned for pickup at [16, 15] and destination E
Arrived at destination with taxi 1
Updated customer position to: [18, 9]
All requests have been processed.
```

Comandos de la central

Vamos a realizar algunas pruebas con los comandos para ver que funcionan adecuadamente.

Si yo decido cambiar el destino del taxi:

```
Opciones de comando para taxis:
1. Parar taxi
2. Reanudar taxi
3. Cambiar destino
4. Volver a la base
q. Salir

Elija una opción: Processing taxi request: {'type': 'pedir_taxi', 'customer_id': 'a', 'pickup': 1730378848.7284198}
DEBUG: Added customer a at position [10, 10]
No available taxis for customer a.
Processing taxi request: {'type': 'pedir_taxi', 'customer_id': 'a', 'pickup': [10, 10], 'destination': 2998}
DEBUG: Added customer a at position [10, 10]
No available taxis for customer a.
3
Introduzca el ID del taxi: 1
Introduzca el nuevo destino (formato: x,y): 8,8
Enviando nuevo destino [8,8] al taxi 1
DEBUG: Sent command change_destination to taxi 1 with payload: {'new_destination': [8, 8]}
```

Utilizo el comando 3, elijo el id 1 del taxi y selecciono la coordenada (8,8) que es el nuevo destino del taxi. Veamos el resultado:

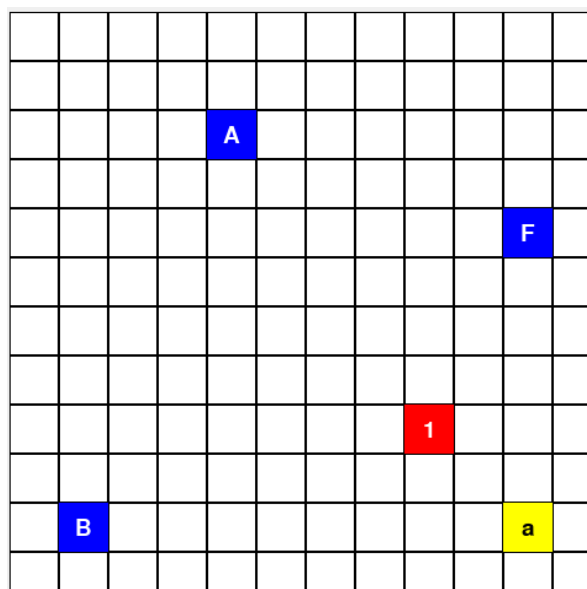
El taxi cambia su estado a RUN y comienza a moverse celda a celda hasta que llega a la posición indicada:

```

Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'KO', 'position': [1, 1]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [1, 1]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [2, 2]}
Moved to position: [2, 2]
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [2, 2]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [3, 3]}
Moved to position: [3, 3]
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [3, 3]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [4, 4]}
Moved to position: [4, 4]
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [4, 4]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [5, 5]}

```

Como podemos observar en ambos mapas, el de la central y el del taxi, el taxi se ha tenido en la posición indicada, y se muestra de color rojo.



Si le indico que vuelva a base.

Opciones de comando para taxis:

1. Parar taxi
2. Reanudar taxi
3. Cambiar destino
4. Volver a la base
- q. Salir

Elija una opción: 4

Introduzca el ID del taxi: 1

Enviando comando de retorno a base al taxi 1

DEBUG: Sent command return_to_base to taxi 1 with payload: None

```
Processing instruction for taxi 1: {'taxi_id': '1', 'command': 'return_to_base', 'payload': None}
```

El taxi procesa la instrucción y vuelve a la base (1,1)

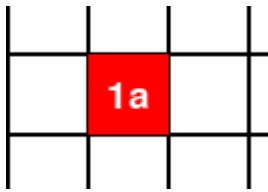
```
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [5, 5]}
Moved to position: [5, 5]
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [5, 5]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [4, 4]}
Moved to position: [4, 4]
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [4, 4]}
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [3, 3]}
Moved to position: [3, 3]
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [2, 2]}
Moved to position: [2, 2]
Map updated - Size: 20x20
Taxis: 1, Customers: 1
Received sensor status: OK
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [2, 2]}
Sending status update: {'type': 'estado_taxi', 'taxi_id': '1', 'status': 'RUN', 'position': [1, 1]}
Moved to position: [1, 1]
Reached destination: [1, 1]
```

El resto de instrucciones como Stop o Resume, funcionan correctamente, como las que he mostrado.

Incidencias

Vamos a realizar una prueba para ver como funcionan las incidencias. El taxi se estaba moviendo al destino con el cliente A. Activamos una incidencia como se observa a continuación, se envía un KO y el taxi se detiene.

```
Enter 'i' to activate incident, 'q' to deactivate incident, or 'esc' to exit: iSent: <STX>OK<ETX>
Incident activated: Sending KO
```

Si desactivamos la incidencia se envía un OK y el taxi continua su camino hacia el destino.

```
q
Incident deactivated: Sending OK
Enter 'i' to activate incident, 'q' to deactivate incident, or 'esc' to exit: Sent: <STX>OK<ETX>
```

Resiliencia

Si realizamos un cierre de la central, esta informa al taxi y al cliente de lo sucedido, y estos siguen funcionando adecuadamente. Cuando la reconectamos también sigue funcionando adecuadamente. Lo mismo sucede con el Sensor y el Cliente. La única resiliencia que no está implementada es la de la clase Digital Engine, porque si la cerramos, los otros componentes no siguen funcionando adecuadamente. Aquí mostramos un ejemplo:

```
C:\UNIVERSIDAD\SD\Práctica\P2\Practical1-SD>python EC_Costumer.py localhost:
Customer a starting at position [10, 10]...
Sent request for pickup at [10, 10] and destination A
Sent request for pickup at [10, 10] and destination A
Request denied: No taxis available
Waiting 4 seconds before retrying...
Sent request for pickup at [10, 10] and destination A
Request accepted: Taxi 1 assigned for pickup at [10, 10] and destination A
Central shutdown detected. Waiting to reconnect...
Reenviando la última solicitud...
Sent request for pickup at [10, 10] and destination A
Sent request for pickup at [10, 10] and destination A
```

Como podemos observar si cerramos la central, esta informa al cliente, y el cliente sigue enviando la misma petición hasta que la reconectamos y entonces continúa la ejecución normal como se observa a continuación.

```
Sent request for pickup at [10, 10] and destination A
Sent request for pickup at [10, 10] and destination A
Sent request for pickup at [10, 10] and destination A
Sent request for pickup at [10, 10] and destination A
Request accepted: Taxi 1 assigned for pickup at [10, 10] and destination A
```

La request es aceptada y la central le asigna el taxi 1 para recoger al cliente y llevarlo al destino A.

Registro

Ahora realizaremos otra prueba para mostrar el funcionamiento de la nueva clase implementada EC_Register. A continuación podemos ver como se registra un taxi.

The screenshot displays the 'EC REGISTRY' interface. It features two main sections. The first section, titled 'Registrar Taxi', contains a 'Taxi ID' input field with the value '9' and a green 'Registrar Taxi' button. The second section, titled 'Dar de Baja Taxi', contains a 'Taxi ID' input field with the placeholder text 'Enter Taxi ID' and a red 'Dar de Baja' button.

```
{"message": "Taxi 9 registered successfully"}
```

Vemos que si introducimos un taxi que ya está registrado nos sale un mensaje de error, indicándonos que ese taxi ya está registrado.

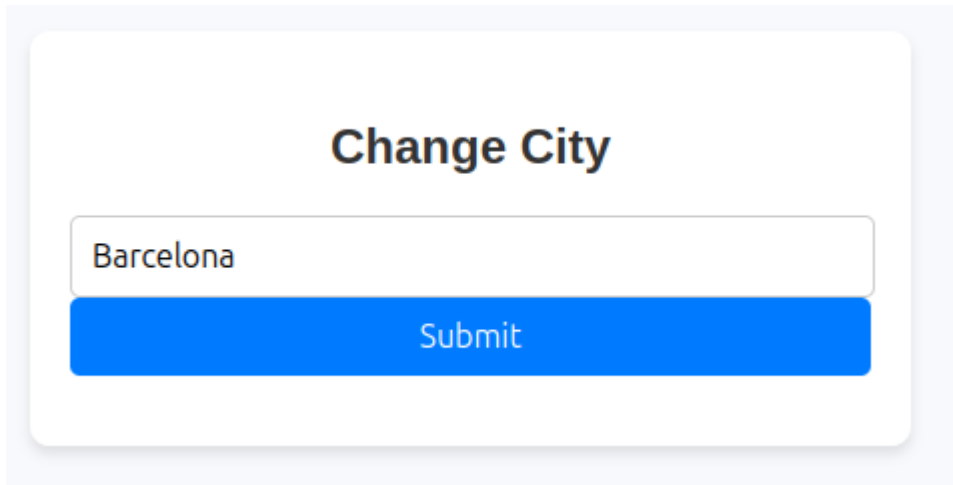
```
{"error": "Taxi already registered"}
```

Ahora eliminaremos el taxi previamente registrado.

```
{"message": "Taxi 9 deleted successfully"}
```

Consulta de tráfico

Desde el front podremos modificar la ciudad:

A screenshot of a web form titled "Change City". It features a text input field containing the word "Barcelona" and a blue "Submit" button below it. The form is centered on a light blue background.

Auditoría

La auditoría guarda distintos eventos que se han producido en un archivo audit.log que se borra al comenzar la ejecución y después se vuelve a crear. Aquí podemos observar un ejemplo del funcionamiento.

```
2024-12-19 19:38:36 - <BrokerConnection node_id=0 host=localhost:9092 <connecting> [IPv4 ('127.0.0.1', 9092)]>:
Connection complete.
2024-12-19 19:38:36 - <BrokerConnection node_id=bootstrap-0 host=localhost:9092 <connected> [IPv4 ('127.0.0.1',
9092)]>: Closing connection.
2024-12-19 19:38:36 - <BrokerConnection node_id=0 host=localhost:9092 <connecting> [IPv4 ('127.0.0.1', 9092)]>:
Connection complete.
2024-12-19 19:38:36 - <BrokerConnection node_id=bootstrap-0 host=localhost:9092 <connected> [IPv4 ('127.0.0.1',
9092)]>: Closing connection.
2024-12-19 19:38:37 - {'timestamp': '2024-12-19T19:38:37.651155', 'ip_address': '127.0.1.1', 'event_type': 'Taxi
Authenticated', 'details': {'taxi_id': '1', 'status': 'END', 'position': [10, 10]}}
2024-12-19 19:38:37 - {'timestamp': '2024-12-19T19:38:37.651327', 'ip_address': '127.0.1.1', 'event_type': 'Taxi
Status Update', 'details': {'taxi_id': '1', 'status': 'END', 'position': [10, 10], 'previous_position': None}}
2024-12-19 19:38:37 - {'timestamp': '2024-12-19T19:38:37.653123', 'ip_address': '127.0.1.1', 'event_type': 'Taxi
Status Update Processed', 'details': {'status_data': {'type': 'estado_taxi', 'taxi_id': '1', 'encrypted_status':
'zPi2mXaevC9pSfSNZ00QyQ==', 'iv_status': '9dPB8w-RWEQKbmNwAlljog==', 'encrypted_position': 'arJABtqxfrnJq1wg-n6fzw==',
'iv_position': 'lg-9SCul6ja4uVf0BpGaHw==', 'encrypted_token': 'eYbzsjue_06wBg0a7FMXp5_9fb7b-
sMw9frCdYoUKkoOnOybJW0_L9Uimcp8TbUn', 'iv_token': '5blR0gX9YhCbEp4Ch18ymw=='}}}}
2024-12-19 19:38:38 - {'timestamp': '2024-12-19T19:38:38.657136', 'ip_address': '127.0.1.1', 'event_type': 'Taxi
Authenticated', 'details': {'taxi_id': '1', 'status': 'END', 'position': [10, 10]}}
2024-12-19 19:38:38 - {'timestamp': '2024-12-19T19:38:38.657377', 'ip_address': '127.0.1.1', 'event_type': 'Taxi
Status Update', 'details': {'taxi_id': '1', 'status': 'END', 'position': [10, 10], 'previous_position': [10, 10]}}
2024-12-19 19:38:38 - {'timestamp': '2024-12-19T19:38:38.673554', 'ip_address': '127.0.1.1', 'event_type': 'Taxi
Status Update Processed', 'details': {'status_data': {'type': 'estado_taxi', 'taxi_id': '1', 'encrypted_status':
```

Vemos como registra la fecha, el día y la hora, el tipo de evento, los datos del taxi y los detalles. Como se puede ver en la captura, los mensajes están cifrados y así aparecen en los logs, para evitar posibles ataques.

Conclusión

Hemos aprendido a desarrollar una aplicación distribuida con la utilización de kafka y sockets para la comunicación. También hemos mejorado en el desarrollo de código y el uso de python, así como en el manejo de errores. Hemos aprendido a conectar estructuras HTML sencillas con API Rest, hemos aprendido a utilizar certificado simétrico y los tokens para una mejor seguridad que hasta ahora ignorábamos y hemos aprendido a manejar una base de datos de una forma mucho más seguida.

En general hemos aprendido cómo conectar las diferentes partes de la aplicación y usarlas en conjunto para poder formar un sistema distribuido de taxis autónomos.