SISTEMAS DISTRIBUIDOS
RELEASE 2

Alejandro García Gil 77042008N & Ginés Caballero Guijarro 54639204J

ÍNDICE

EC_CTC

DESCRIPCIÓN DEL ARCHIVO:

El archivo EC_CTC.py es una simulación que combina funcionalidades de servidor web, consultas climáticas y monitoreo del estado del tráfico en una ciudad específica. Esta aplicación permite cambiar la ciudad monitoreada, verificar el estado del tráfico y consultar la temperatura actual a través de un servidor Flask, integrando también un menú interactivo en la línea de comandos para facilitar la interacción.

FUNCIONALIDADES CLAVE:

1. Consultas climáticas y estado del tráfico:

- **`get_current_temperature(city):`**
  Obtiene la temperatura actual de una ciudad especificada usando la API de OpenWeather. Si ocurre un error durante la consulta, se registra en el archivo de logs.

- **`determine_traffic_status(temp):`**
  Calcula el estado del tráfico basado en un umbral de temperatura (`TEMPERATURE_THRESHOLD`). Si la temperatura es igual o superior al umbral, el tráfico es clasificado como "OK"; de lo contrario, se clasifica como "KO".

2. Servidor Flask para endpoints HTTP:

- **`/get_traffic_status:`**
  Devuelve el estado actual del tráfico, la ciudad monitoreada y un timestamp en formato JSON.
- **`/set_city:`**
  Cambia la ciudad actual y actualiza el estado del tráfico en función de la nueva ubicación. Requiere un cuerpo JSON con la clave `city`.
- **`/get_temperature:`**
  Devuelve la temperatura actual de la ciudad monitoreada en formato JSON.
- **`/shutdown:`**
  Detiene el servidor Flask mediante una función del entorno de werkzeug.

3. Menú interactivo en CLI:

- **`cli_menu():`**
  Proporciona una interfaz interactiva basada en PyInquirer, permitiendo al usuario:
  - Cambiar la ciudad monitoreada.
  - Ver el estado actual del tráfico.
  - Consultar la temperatura actual.
    Incluye validaciones y maneja errores de conexión con el servidor Flask.

4. Tareas periódicas en segundo plano:

- **`periodic_temperature_check()`:**
  Cada 10 segundos, consulta la temperatura actual y actualiza el estado del tráfico si detecta un cambio, registrando estos eventos en los logs.

5. Gestión de logs:

El archivo utiliza el módulo `logging` para registrar eventos clave, incluyendo cambios de ciudad, errores en consultas a la API y cambios en el estado del tráfico. Los logs se almacenan en el archivo `LOGS/ec_ctc.log`.

FLUJO DE DATOS:

Inicio:
Al ejecutar el script, se inicia un servidor Flask en segundo plano y se lanza un hilo que actualiza periódicamente el estado del tráfico.

Interacción del usuario:
El menú interactivo permite cambiar la ciudad monitoreada o consultar el estado actual. Las interacciones son gestionadas a través de solicitudes HTTP hacia el servidor Flask.

Procesamiento de datos climáticos:
La temperatura de la ciudad es obtenida mediante la API de OpenWeather y utilizada para determinar el estado del tráfico.

Actualización del estado del tráfico:
Si la temperatura cambia y cruza el umbral configurado, el estado del tráfico se actualiza y se registra en los logs.

EC_CENTRAL

DESCRIPCIÓN DEL ARCHIVO:

El archivo EC_Central.py representa el núcleo de una central que coordina y gestiona las operaciones de taxis y clientes, integrando comunicación mediante Kafka y proporcionando interfaces web para la visualización de datos. Este archivo incluye lógica para monitorear el estado del tráfico, gestionar actualizaciones de taxis y procesar solicitudes de clientes. En este Release nos hemos centrado en el desarrollo de la central para integrarla con un API con la finalidad de trabajar desde un front en web. Además, se ha desarrollado un pequeño sistema de logs con la finalidad de monitorear mejor el funcionamiento del proyecto

FUNCIONALIDADES CLAVE (SEGUNDO RELEASE):

1. Configuración de logs

- **Propósito:** Configura el sistema de registro para capturar eventos importantes, como cambios de estado, errores o actualizaciones de tráfico.
- **Detalles:**
  - Los registros se almacenan en el archivo LOGS/central.log.
  - Incluyen el nivel de registro (INFO, WARNING, ERROR) y el formato con fecha y hora en formato ISO 8601.
- **Uso:** Facilita la monitorización y depuración del sistema.

2. Creación del servidor Flask

- El servidor web Flask actúa como interfaz para exponer datos y funcionalidades clave mediante endpoints RESTful.
- Permite la interacción entre el sistema central y otros componentes, como la interfaz gráfica o servicios externos.

3. Endpoints RESTful principales

- **/**
  Renderiza la página principal (map.html) con una representación gráfica del sistema.

- **/get_taxis**
  Devuelve información de los taxis desde un archivo local (taxis.json), proporcionando datos esenciales para la visualización.

- **/get_map**
  Devuelve el estado del mapa (mapa.json), incluyendo ubicaciones de taxis, clientes y destinos.

- **/get_all_taxis**
  Recupera los datos actuales de todos los taxis. Si el archivo no existe, devuelve una lista vacía.

- **/update_taxis**
  Permite actualizar la información de los taxis a través de una solicitud POST. Valida el formato del JSON y registra errores en caso de problemas de escritura.

4. Gestión del estado del tráfico

- **get_traffic_status:**
  Consulta el estado del tráfico desde un servicio externo (CTC) mediante una solicitud HTTP GET. Registra los resultados o errores en los logs.

- **periodic_city_status:**
  Monitorea continuamente el estado del tráfico y ejecuta acciones automáticas en función de los cambios detectados:
  - Si el tráfico cambia de "OK" a "KO", se envía a los taxis de vuelta a la base.
  - Si el tráfico cambia de "KO" a "OK", se reanudan las operaciones.

5. Gestión de taxis

- **return_taxis_to_base:**
  Envía un comando a los taxis para que regresen a la base en caso de tráfico adverso.

- **resume_taxi_operations:**
  Reanuda las operaciones normales de los taxis cuando el tráfico mejora.

FLUJO DE DATOS:

1. Inicio del sistema

- La aplicación se inicia desde la línea de comandos, configurando las conexiones necesarias con Kafka, cargando datos iniciales de taxis y mapa, y arrancando múltiples hilos de ejecución:
    - Un servidor Flask para gestionar las peticiones web.
    - Hilos para escuchar actualizaciones de Kafka.
    - Un hilo para monitorear periódicamente el estado del tráfico.

2. Interacción del usuario

- Los usuarios pueden interactuar con el sistema a través de la interfaz gráfica (map.html) o mediante solicitudes REST hacia los endpoints del servidor Flask.

3. Gestión del tráfico

- El estado del tráfico es consultado cada 10 segundos mediante la función periodic_city_status.
- Si se detectan cambios significativos, se toman acciones automáticas:
    - Regresar los taxis a la base si el tráfico empeora.
    - Reanudar operaciones si el tráfico mejora.

4. Respuesta a eventos de Kafka

- **Actualizaciones de taxis:**
  Kafka proporciona información sobre el estado y posición de los taxis. Estos datos se procesan y actualizan en los archivos locales y en el mapa.
- **Solicitudes de clientes:**
  Las solicitudes de servicio son procesadas desde Kafka, verificando la disponibilidad de taxis y asignándolos a los clientes.

5. Visualización en tiempo real

- Los datos de taxis y mapa son visualizados en tiempo real en la interfaz gráfica, permitiendo al usuario ver el estado actual del sistema.

EC_DE

DESCRIPCIÓN DEL ARCHIVO (SEGUNDO RELEASE):

La clase DigitalEngine gestiona el comportamiento de un taxi. Este archivo representa la lógica de cada taxi, simulando su comportamiento al recibir asignaciones de clientes, desplazarse hacia el cliente, y luego llevar al cliente al destino deseado. DigitalEngine es responsable de gestionar la posición del taxi, así como su estado operativo. En este release nos hemos limitado a añadir un simple menú con pyInquirer con el que dar de alta o de baja el taxi en la base de datos. Además, hemos mejorado la seguridad entre los taxis y la central y la API de registro a través de distintos métodos de encriptación.

FUNCIONALIDADES CLAVE:

1. Configuración del entorno

- **load_dotenv**
  Carga variables de configuración desde un archivo .env, facilitando la separación entre el código fuente y las configuraciones específicas del entorno.

2. Creación de un contexto SSL

- **create_ssl_context**
  Genera un contexto SSL que valida las comunicaciones seguras mediante un certificado. Esto garantiza la seguridad y autenticidad de las conexiones con el servidor de registro.

3. Registro y desregistro de taxis

- **register_taxi**
  Registra un taxi en el sistema utilizando una conexión segura validada por SSL.
- **deregister_taxi**
  Desregistra un taxi del sistema de manera similar, eliminándolo de la lista activa de taxis en la central.

4. Menú interactivo para registro

- **show_registry_menu**
  Presenta un menú interactivo que permite al usuario elegir entre registrar, desregistrar o continuar sin realizar cambios en el registro del taxi.

5. Gestión del estado operativo

- **resume_operations**
  Reanuda las operaciones normales del taxi tras recibir un comando de la central. Cambia el estado del taxi a "OK" y lo marca como disponible para asignaciones.

6. Regreso a la base

- **return_to_base**
  Configura al taxi para regresar a su posición base cuando recibe el comando correspondiente. Marca al taxi como no disponible durante este proceso.
- **monitor_return_to_base**
  Supervisa el proceso de regreso a la base, verificando periódicamente si el taxi ha alcanzado su posición objetivo. Una vez allí, cambia su estado a "KO" y detiene la supervisión.

FLUJO DE DATOS:

1. Inicio del sistema

- Se cargan las configuraciones desde el archivo .env, incluyendo información sensible como certificados y rutas.
- El menú interactivo permite al usuario registrar o desregistrar un taxi de forma segura antes de iniciar las operaciones.

2. Interacción con la central

- El taxi responde a comandos operativos como RESUME_OPERATIONS y RETURN_TO_BASE, enviados por la central a través de Kafka.
- Los cambios en el estado y la posición del taxi se reflejan en tiempo real en el sistema.

3. Supervisión del estado

- Si el taxi debe regresar a la base, inicia el proceso y lo supervisa hasta completar la acción, actualizando su estado al llegar.

4. Seguridad

- Todas las comunicaciones con el servidor de registro utilizan conexiones SSL seguras, validadas con certificados, para garantizar la autenticidad y seguridad de los datos transmitidos.

EC_REGISTRY

DESCRIPCIÓN DEL ARCHIVO

El archivo EC_Registry.py implementa un servidor Flask que actúa como registro de taxis en un sistema de gestión. Su principal responsabilidad es manejar el registro, desregistro y verificación de taxis, sirviendo como intermediario entre los componentes centrales del sistema. El servidor asegura la comunicación mediante certificados SSL generados automáticamente.

FUNCIONALIDADES CLAVE

1. Generación de certificados

- **generate_certificates**
  Genera un certificado autofirmado y una clave privada si no existen previamente. Estos se utilizan para garantizar comunicaciones seguras mediante HTTPS:
  - Incluye información de la organización y el servidor.
  - Guarda los archivos cert.pem y key.pem para su uso por el servidor Flask.

2. Gestión de taxis desde la central

- **get_taxis_from_central**
  Recupera la lista actual de taxis desde el servidor central utilizando su API. Permite al registro sincronizarse con la base de datos central.
- **update_taxis_in_central**
  Envía actualizaciones de la lista de taxis al servidor central. Garantiza que los cambios realizados en el registro se reflejen en la base de datos central.

3. Endpoints para la gestión de taxis

- **register_taxi**
  Permite registrar un taxi en el sistema. Verifica si el taxi ya está registrado y, de no estarlo, lo agrega a la lista:
  - Configura el taxi con coordenadas iniciales y estado predeterminado.
  - Actualiza la base de datos central con el nuevo registro.
- **deregister_taxi**
  Elimina un taxi del sistema. Verifica si el taxi existe antes de eliminarlo y actualiza la central para reflejar el cambio.
- **is_registered**
  Comprueba si un taxi está registrado en el sistema. Devuelve un estado HTTP 200 si el taxi está registrado, o 404 si no lo está.

FLUJO DE DATOS

1. Inicio del servidor

- El servidor se inicia con tres argumentos:
    1. Dirección IP de la central.
    2. Puerto de la central.
    3. Puerto del registro.
- Antes de arrancar, genera los certificados SSL necesarios si no existen.

2. Registro y desregistro de taxis

- Los taxis pueden registrarse o desregistrarse mediante los endpoints:
    - **Registro:** El taxi se agrega a la lista local y se sincroniza con la central.
    - **Desregistro:** Se elimina el taxi de la lista local y de la central.

3. Sincronización con la central

- Cada acción relacionada con los taxis se sincroniza con la base de datos central mediante las funciones get_taxis_from_central y update_taxis_in_central.

4. Comunicaciones seguras

- Todas las interacciones con el registro utilizan HTTPS, aseguradas mediante los certificados generados.

DESPLIEGUE DE LA APLICACIÓN

Para desplegar nuestra práctica en los tres ordenadores de la universidad, seguimos un proceso estructurado que asegura la correcta interacción entre losdistintos componentes del sistema. Aquí describimos los pasos en detalle:

Ordenador 1: Configuración de Kafka y Clientes

1. **Instalación de Kafka:**

→ Primero, instalamos Kafka en el ordenador, lo que incluye configurar Zookeeper yel servidor de Kafka. Esto es fundamental ya que Kafka actúa como el mediador de mensajes entre los diferentes componentes de nuestra práctica.

→ Modificamos el archivo server.properties de Kafka, descomentando laslíneas necesarias para activar configuraciones específicas que mejoren el rendimiento o la seguridad según nuestras necesidades.

2. **Levantar Kafka:**

→ Iniciamos Zookeeper y el servidor de Kafka para asegurarnos de que el sistema de mensajería esté operativo antes de iniciar cualquier componente que dependa deél.

3. **Levantar la central y el front:**

→ En el segundo ordenador, solo levantamos la aplicación de la central (central). Esta aplicación es crucial ya que procesa todas las solicitudes y comandos enviadospor los clientes a través de Kafka y toma decisiones correspondientes sobre cómo manejar los taxis. Abriremos el front para enseñar el gráfico con los taxis.

Ordenador 2: Configuración de la Central

1. **Levantar la CTC:**

→ En el segundo ordenador, solo levantamos la aplicación de la CTC (central). Esta aplicación es importante ya que actualiza el estado de la ciudad para saber si es circulable o no.

Ordenador 3: Configuración de los Taxis

Instalación de Dependencias:

→ Aseguramos que todas las librerías necesarias, como las de Confluent Kafka para la integración con Kafka y Matplotlib para cualquier visualización necesaria,estén instaladas y configuradas correctamente.

Levantar los Taxis y el Registry:

→ Levantamos la API de registry, la cual nos dará la posibilidad de registrar, dar de baja y confirmar si están los taxis en la bd.

→ Levantamos dos instancias de Digital Engine, cada uno representando un taxi. Cada Digital Engine incluye un proceso que simula los sensores del taxi,enviando y recibiendo mensajes a través de Kafka para actualizar su estado y posición según las instrucciones de la central.

Regreso al Ordenador 2

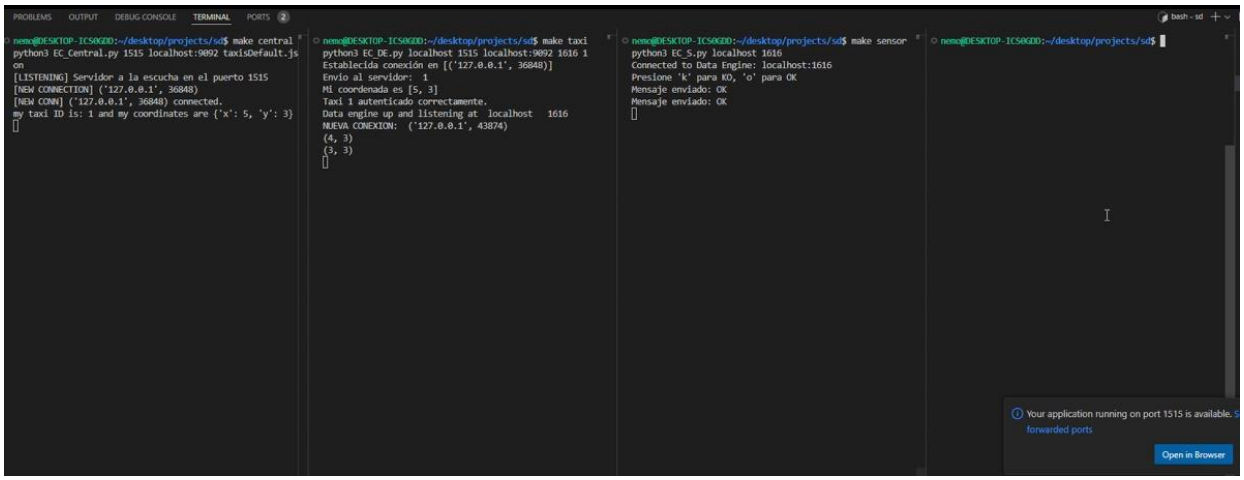
Activar los Clientes:

→ Finalmente, volvemos al primer ordenador para activar y monitorear los clientes.A este punto, todos los componentes están ya operativos y los clientes pueden comenzar a interactuar con el sistema, enviando solicitudes para servicios de taxi yrecibiendo respuestas.

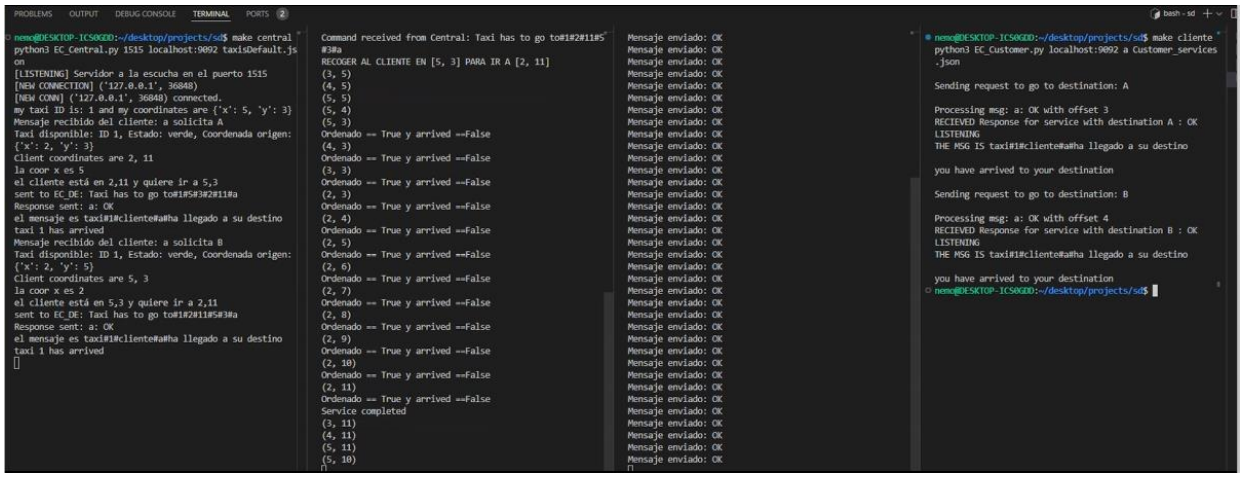
Para la ejecución se ha creado tanto un makefile como un .bat para la ejecución de Kafka y ejecución de los scripts para tener un poco mas de posibilidades de despliegue, el makefile usa apache do1ckerizado.

EJEMPLO DE USO Y EJECUCIÓN:

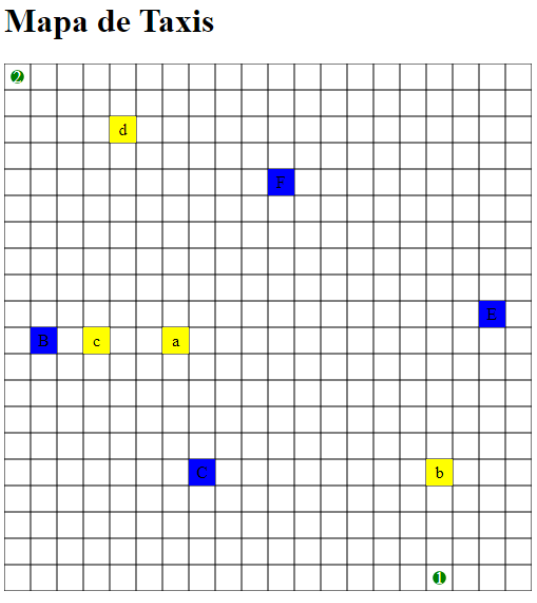
Iniciamos la central y añadimos un taxi junto a su sensor:



En esta imagen, ya se ha ejecutado el servicio de un cliente y se ha completado:



El mapa se ve de la siguiente manera:



El registry se ve de la siguiente manera:

```
PS C:\Users\Ginés\Desktop\SD-2425> python .\EC_Registry.py 192.168.23.1 5001 5002
registry API working on 192.168.23.1:5002 sending and getting bd data from central with API at http://192.168.23.1:5001

 * Serving Flask app 'EC_Registry'
 * Debug mode: off
```

Y el CTC se ve de la siguiente manera:

```
PS C:\Users\Ginés\Desktop\SD-2425> python .\EC_CTC.py 6761
Iniciando EC_CTC... en 192.168.23.1:6761
 * Serving Flask app 'EC_CTC'
 * Debug mode: off
? Selecciona una acción:  (Use arrow keys)
 ❯ Cambiar ciudad
   Ver estado actual del tráfico
   Ver temperatura actual
   Salir
```