DESPLIEGUE PRÁCTICA 1

EASYCAB

Grupo 2

Sistemas distribuidos

Curso 2024/2025

ERARDO ALDANA PESSOA. 26960240P

JORGE PAREDES SOLER

INGENIERÍA INFORMÁTICA

UNIVERSIDAD DE ALICANTE

DESPLIEGUE SD:

Repositorio Github: https://github.com/eap59-ua/easycabSD

Nota: cambiar localhost por IP Destino (en la misma red)

Orden de inicio

Zookeeper → Kafka → DB → Topics → Central → Taxis → Sensores → Clientes

T1 CENTRAL:

```
python EC_Central.py 5000 localhost:9092

        # Sintaxis: python EC_Central.py <puerto_escucha> <kafka_broker> [--db-host <host>] [--db-port <puerto>]

python EC_Central.py 5000 localhost:9092 --db-host localhost --db-port 5432
```

T2 CUSTOMER:

```
# Sintaxis: python EC_Customer.py <kafka_broker> <client_id>

# Para el cliente 'a'

python EC_Customer.py localhost:9092 a


# Para el cliente 'b' (en otra terminal)

python EC_Customer.py localhost:9092 b
```

T3 TAXI EC_DE

```
# PARA EL TAXI 1 (DIGITAL ENGINE)

# Sintaxis: python EC_DE.py <central_ip> <central_port> <kafka_broker> <sensor_port> <taxi_id>

python EC_DE.py localhost 5000 localhost:9092 5001 1

# PARA EL SENSOR DEL TAXI 1

# Sintaxis: python EC_S.py <de_ip> <de_port>

python EC_S.py localhost 5001

# PARA EL TAXI 2 (EN OTRA TERMINAL)

python EC_DE.py localhost 5000 localhost:9092 5002 2

# PARA EL SENSOR DEL TAXI 2 (EN OTRA TERMINAL)

python EC_S.py localhost 5002
```

```
# TERMINAL 1
```

```
CD ~/KAFKA
BIN/ZOOKEEPER-SERVER-START.SH CONFIG/ZOOKEEPER.PROPERTIES
```

```
# TERMINAL 2
```

```
CD ~/KAFKA
BIN/KAFKA-SERVER-START.SH CONFIG/SERVER.PROPERTIES
```

Para no tener las 2 terminales abiertas:

```
# CONFIGURAR COMO SERVICIOS DEL SISTEMA
```

sudo systemctl start zookeeper

sudo systemctl start kafka

```
# VERIFICAR ESTADO
```

sudo systemctl status zookeeper

sudo systemctl status Kafka

```
# PARA QUE INICIEN AUTOMÁTICAMENTE AL ARRANCAR EL SISTEMA
```

sudo systemctl enable zookeeper

sudo systemctl enable Kafka

USO DE DOCKER COMPOSE:

PARA PC1(CENTRAL)

```
services:

 postgres:

  image: postgres:latest

  environment:

   POSTGRES_USER: postgres

   POSTGRES_PASSWORD: postgres

   POSTGRES_DB: central_db

  ports:

   - "5432:5432"
```

PARA PC2(CUSTOMER)

```
services:

 zookeeper:

  image: confluentinc/cp-zookeeper:latest

  environment:

   ZOOKEEPER_CLIENT_PORT: 2181


 kafka:

  image: confluentinc/cp-kafka:latest

  depends_on:

   - zookeeper

  ports:

   - "9092:9092"
```

PARA PC3: (TAXIS)

```
version: '3'

services:

 taxi1:

  build:

   context: ./taxi

   dockerfile: Dockerfile

  environment:

   - TAXI_ID=1

   - CENTRAL_IP=<ip_pc2>

   - CENTRAL_PORT=<puerto_central>

   - KAFKA_BROKER=<ip_pc2>:9092

  command: python EC_DE.py ${CENTRAL_IP}:${CENTRAL_PORT} ${KAFKA_BROKER}
${SENSOR_PORT} ${TAXI_ID}


 sensor1:

  build:

   context: ./sensor

   dockerfile: Dockerfile

  environment:

   - TAXI_ID=1

   - DE_PORT=<puerto_de>

  command: python EC_S.py localhost:${DE_PORT}

  depends_on:

   - taxi1


 # Puedes replicar para más taxis

 taxi2:

  build:
```

```
    context: ./taxi
    dockerfile: Dockerfile
  environment:
    - TAXI_ID=2
    - CENTRAL_IP=<ip_pc2>
    - CENTRAL_PORT=<puerto_central>
    - KAFKA_BROKER=<ip_pc2>:9092
  command: python EC_DE.py ${CENTRAL_IP}:${CENTRAL_PORT} ${KAFKA_BROKER}
${SENSOR_PORT} ${TAXI_ID}


 sensor2:
  build:
    context: ./sensor
    dockerfile: Dockerfile
  environment:
    - TAXI_ID=2
    - DE_PORT=<puerto_de>
  command: python EC_S.py localhost:${DE_PORT}
  depends_on:
    - taxi2´
```

FUNCIONAMIENTO DESCONEXIÓN DE TAXI

1. Si el taxi se desconecta:
    - La central inicia un timer de 10 segundos
    - Durante esos 10 segundos:
        - Si el taxi vuelve a conectarse, puede retomar su servicio desde donde estaba
        - El servicio se mantiene "en espera"
    - Después de 10 segundos:
        - Se marca como incidencia
        - Se notifica al cliente
        - El servicio se cancela completamente
2. Si intenta reconectarse:
    - Antes de 10 segundos:
        - Se le permite retomar el servicio desde su última posición
        - Mantiene su cliente asignado
    - Después de 10 segundos:
        - Debe empezar desde [1,1]
        - Debe autenticarse como nuevo taxi
        - No puede retomar servicios antiguos

FUNCIONAMIENTO GENERAL

CLIENTE -> SOLICITA SERVICIO -> CENTRAL

CENTRAL -> BUSCA TAXI LIBRE -> ASIGNA TAXI

CENTRAL -> NOTIFICA CLIENTE (OK/KO)

CENTRAL -> ENVÍA ORDEN AL TAXI PARA RECOGER CLIENTE

TAXI -> SE MUEVE A POSICIÓN DEL CLIENTE

TAXI -> RECOGE CLIENTE

TAXI -> LLEVA CLIENTE A DESTINO

CENTRAL -> NOTIFICA CLIENTE QUE LLEGÓ A DESTINO

CLIENTE -> ESPERA 4S -> SOLICITA NUEVO SERVICIO (SI TIENE)

DESPLIEGUE EN 3 ORDENADORES(FÍSICAS/VIRTUALES)

PORTÁTIL 1:

└── **VirtualBox**

  ├── **VM1 (Linux)**

  │ └── **EC_Central**

  └── **VM2 (Linux) / Windows host -> Docker -> Taxis (EC_DE & EC_S)**

    └── **Docker**

      └── **Taxis (EC_DE & EC_S)**

PORTÁTIL 2:

└── **VirtualBox**

  └── **VM3 (Linux)**

    ├── **Kafka**

    └── **EC_Customer**

Esta configuración es más correcta porque:

1. **Cumple con el diagrama de la documentación (Figura 4) donde muestra:**
    - **PC1: EC_Customer + Kafka**
    - **PC2: EC_Central**
    - **PC3: EC_DE & EC_S**
2. **Distribución de componentes:**
    - **VM1: Central de control**
    - **VM2: Taxis en contenedores Docker**
    - **VM3: Clientes + Kafka**

CONFRIGURACION DE LAS IPS

 # VM1 (Central)

IP: 192.168.100.10

 # VM2 (Docker/Taxis)

IP: 192.168.100.20

 # VM3 (Customers + Kafka) en Portátil 2

IP: 192.168.100.30

PASOS PARA PROBAR LA APLICACIÓN EN UN SOLO PORTÁTIL

1. **INSTALACIÓN DE KAFKA Y ZOOKEEPER**:
    - PRIMERO, ASEGÚRATE DE TENER KAFKA Y ZOOKEEPER INSTALADOS EN TU PORTÁTIL. PUEDES HACERLO DESDE UN TERMINAL. SI NO LO HAS HECHO, SIGUE LA GUÍA OFICIAL DE KAFKA PARA INSTALARLO Y EJECUTARLO.
2. **INICIAR ZOOKEEPER**:
    - KAFKA NECESITA ZOOKEEPER PARA FUNCIONAR. ABRE UNA TERMINAL Y EJECUTA EL SIGUIENTE COMANDO PARA INICIAR ZOOKEEPER:

- BIN/ZOOKEEPER-SERVER-START.SH CONFIG/ZOOKEEPER.PROPERTIES

3. **INICIAR KAFKA**:

- EN OTRA TERMINAL, INICIA EL SERVIDOR DE KAFKA:

BIN/KAFKA-SERVER-START.SH CONFIG/SERVER.PROPERTIES

ABRE TRES TERMINALES, UNA PARA CADA APLICACIÓN.

3.1 EN LA PRIMERA TERMINAL, EJECUTA LA APLICACIÓN CENTRAL:

CD EASYCAB/CENTRAL

PYTHON CENTRAL_APP.PY <PUERTO_ESCUCHA> <IP_BROKER> <PUERTO_BD>

NOTAS:

- Traer hoja rellenada, impresa

https://www.cherryservers.com/blog/install-mongodb-ubuntu-22-04

LA MANERA DE SABER LA POSICIÓN DEL CLIENTE, EL CLIENTE TIENE QUE ELEGIR DONDE ESTA. SE ABRE EL SERVIDOR, SE CONECTA EL CLIENTE Y EMPIEZA LA COMUNICACIÓN INICIAL PARA:

- el servidor pregunta dónde está el cliente

- Servidor: Saber la posición del cliente, coordenadas

- EL servidor recibe esa coordenada, la acepta, asigna taxi, etc...

HACERLO EN LINUX

Instalación y uso de Kafka en Linux

1. Instalación

1. Asegúrate de tener Java instalado:

sudo apt update

sudo apt install default-jdk

2. Descarga Kafka:

wget https://downloads.apache.org/kafka/3.5.1/kafka_2.13-3.5.1.tgz

3. Extrae el archivo:

tar -xzf kafka_2.13-3.5.1.tgz

4. Mueve la carpeta extraída a /opt:

sudo mv kafka_2.13-3.5.1 /opt/kafka

2. Configuración

1. Edita el archivo de configuración de Kafka:

sudo nano /opt/kafka/config/server.properties

2. Asegúrate de que las siguientes líneas estén descomentadas y configuradas:

listeners=PLAINTEXT://your_server_ip:9092

advertised.listeners=PLAINTEXT://your_server_ip:9092

Reemplaza your_server_ip con la IP de tu servidor.

3. Iniciar Kafka

1. Inicia Zookeeper:

/opt/kafka/bin/zookeeper-server-start.sh /opt/kafka/config/zookeeper.properties

2. En otra terminal, inicia el servidor Kafka:

/opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/server.properties

4. Uso básico

1. Crear un topic:

/opt/kafka/bin/kafka-topics.sh --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

2. Iniciar un productor:

/opt/kafka/bin/kafka-console-producer.sh --topic test-topic --bootstrap-server localhost:9092

3. En otra terminal, iniciar un consumidor:

/opt/kafka/bin/kafka-console-consumer.sh --topic test-topic --from-beginning --bootstrap-server localhost:9092

5. Uso en Python

1. Instala la biblioteca kafka-python:

pip install kafka-python

2. Usa el código proporcionado anteriormente para producir y consumir mensajes.

RESUMEN PRACTICA 1 Y KAFKA:

Para esta práctica de *EasyCab*, Kafka será utilizado como un intermediario de mensajes para gestionar la comunicación entre los distintos componentes distribuidos (CENTRAL, taxis y clientes). Kafka actúa como un sistema de *streaming* que permite manejar de manera eficiente los eventos en tiempo real, como las actualizaciones de posiciones de los taxis, solicitudes de servicios, e incidencias en los recorridos.

Te explico cómo Kafka encaja en esta solución:

1. Componentes principales de Kafka

- **Producer**: Es cualquier componente que genera eventos. En este caso, los taxis y los clientes actuarán como *producers* porque enviarán eventos a la CENTRAL.
- **Consumer**: La CENTRAL actuará como un *consumer* porque estará constantemente escuchando los eventos que envían los taxis y los clientes.
- **Topics**: Kafka organiza los eventos en "topics". Podrías crear varios topics como:
    - taxis-position-updates: para las posiciones que los taxis envían mientras se mueven.
    - customers-service-requests: para las solicitudes de los clientes pidiendo un servicio.
    - taxis-status: para los cambios de estado de los taxis (ej. STOP, RESUME, etc.).
    - central-responses: para las respuestas de la CENTRAL tanto a los taxis como a los clientes.

2. Roles en tu aplicación

- **Taxis como Producers**: Cada taxi, a medida que cambia de posición, enviará un mensaje a Kafka con la nueva posición usando un *producer*. También enviará alertas de los sensores en caso de incidencias. Este mensaje se envía al topic taxis-position-updates.

- **Central como Consumer y Producer**: La CENTRAL tendrá consumidores escuchando los topics taxis-position-updates para recibir actualizaciones de los taxis, y customers-service-requests para recibir solicitudes de servicio de los clientes. Además, puede actuar como un *producer*, enviando mensajes al topic taxis-status para notificar a los taxis de cambios en su estado (instrucciones como "parar", "reanudar", "volver a base").

- **Clientes como Producers y Consumers**: Los clientes serán *producers* al enviar solicitudes de servicio, y *consumers* al recibir respuestas de la CENTRAL (OK/KO) para saber si se les ha asignado un taxi.

3. Cómo implementar Kafka en el sistema

- **Configuración inicial**:

    1. **Broker/Bootstrap-server**: Kafka necesita estar desplegado con un *broker* que gestiona los mensajes. Al iniciar la CENTRAL, debes pasarle la IP y puerto de este broker.

    2. **Clientes Kafka**: Los taxis, clientes y CENTRAL usarán clientes de Kafka para enviar y recibir mensajes. Dependiendo del lenguaje que elijas (Java, Python, etc.), existe una API Kafka específica que te permite implementar esto de manera sencilla.

- **Eventos en tiempo real**: Cada vez que el taxi cambia de posición, envía un evento al topic taxis-position-updates. La CENTRAL consume esos eventos y actualiza el mapa. Además, si el taxi se enfrenta a una contingencia (semáforo rojo, peatón cruzando), enviará una alerta que será procesada por la CENTRAL.

- **Distribución de la carga**: Al usar Kafka, la comunicación entre componentes será más eficiente y escalable. Los eventos son distribuidos de manera asíncrona, lo que significa que el sistema no se bloquea esperando respuestas. Kafka se encarga de manejar la persistencia y replicación de los mensajes.

4. Ventajas de Kafka en esta práctica

- **Desacoplamiento**: Los taxis y los clientes no necesitan estar directamente conectados con la CENTRAL. Kafka actúa como intermediario, permitiendo que los componentes se comuniquen de manera asíncrona.

- **Resiliencia**: Si un taxi o cliente se desconecta, los mensajes enviados a Kafka pueden almacenarse y entregarse cuando se reconecten.

- **Escalabilidad**: Kafka es ideal para aplicaciones distribuidas en las que múltiples taxis y clientes están constantemente generando eventos. Kafka permite manejar grandes volúmenes de mensajes sin saturar la red.

5. Pasos sugeridos para implementar Kafka

1. **Instalar y configurar Kafka**:

    - Desplegar un broker de Kafka. Puedes instalarlo en tu máquina local o en una VM.

2. **Desarrollar los clientes Kafka**:

- Implementar *producers* en los taxis y clientes que envíen eventos a Kafka.
- Implementar *consumers* en la CENTRAL que procesen esos eventos.

3. **Crear los topics**:
   - Define los topics necesarios como taxis-position-updates, customers-service-requests, etc.
4. **Pruebas**:
   - Prueba la comunicación entre taxis y la CENTRAL, asegurándote de que Kafka recibe, almacena y distribuye correctamente los eventos.

Esto te permitirá utilizar Kafka de manera efectiva para la simulación de tu sistema distribuido de taxis autónomos, garantizando una comunicación en tiempo real robusta y eficiente.

KAFKA: (LANZAMOS PRIMERO ZOOKEPER, DESPUÉS BROKER)

Soluciones posibles errores en comunicación con Zookeeper y Broker:

- Firewall de Windows, desactivarlo, probar, y después activarlo
- Cerrar las consolas

Verificación de Zookeper activo:

En la consola: netstat (puerto 2181)

PRACTICA 1 Y KAFKA:

Puntuación sobre la práctica, el hecho de matar y volver a levantar los servicios, y que siga funcionando nuestro programa.

Central: productor- consumidor de los clientes y los taxis

Cliente: productor y consumidor con la central, produce peticiones a la central

1 topic será usado por todos los clientes

NO hacer 1 topic por cada cliente

Central ←-------- REQ_CUST ←---------- C1, C2…

Central --------→ ANS_CUST ----------→ C1, C2…

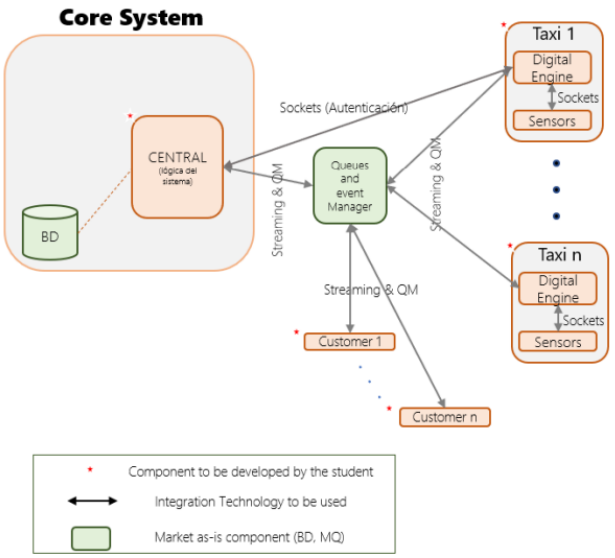
Central ←-------- REQ_SERV ----------→ T1, T2, T3….

T1 (produce)--------→ MOV (1 cola) (Consume) ------→ Central

Nota importante: Lo ideal es usar el mínimo número de Topics para los taxis

Diseño técnico

Se propone al estudiante implementar un sistema distribuido compuesto, al menos, de los siguientes componentes software:



PRACTICA 1 SISTEMAS DISTRIBUIDOS

Estados:

Para lanzar diferentes contenedores y ejcutarlos, utilizaremos Compose.

Practica 1: Python, mongoDB y Docker.

Detalles de cada componente:

1. **Python**:
    - Es perfecto para desarrollar la lógica de la aplicación, con bibliotecas que permiten conectarse a bases de datos, manejar servidores web, etc.
    - La librería pymongo en Python permite interactuar fácilmente con MongoDB.
2. **MongoDB/ PostgreSQL**
    - MongoDB es una base de datos NoSQL que maneja datos en formato JSON/BSON. Es bastante popular para aplicaciones web y proyectos que requieren flexibilidad en el manejo de datos no estructurados.
    - Puedes conectarlo a Python a través de pymongo para almacenar y consultar datos.
3. **Docker**:

    Instalar el DockerHub, monitorizamos todos los contenedores

    https://docs.docker.com/docker-hub/quickstart/

- Docker permite contenedorar aplicaciones, lo que facilita su ejecución en diferentes entornos sin problemas de configuración.
- Puedes ejecutar tanto tu aplicación de Python como MongoDB en contenedores separados, comunicándolos a través de una red interna de Docker.

Flujo de trabajo usando Python, PostgreSQL y Docker:

- **Python** desarrollaría la lógica de negocio de tu aplicación.
- **MongoDB /PostgreSQL** actuaría como el almacén de datos.
- **Docker** te permitiría crear contenedores:
  - Uno para tu aplicación Python.
  - Otro para la base de datos MongoDB.

Además, Docker permite manejar dependencias y versiones, por lo que es una excelente opción si estás buscando un entorno fácilmente reproducible.

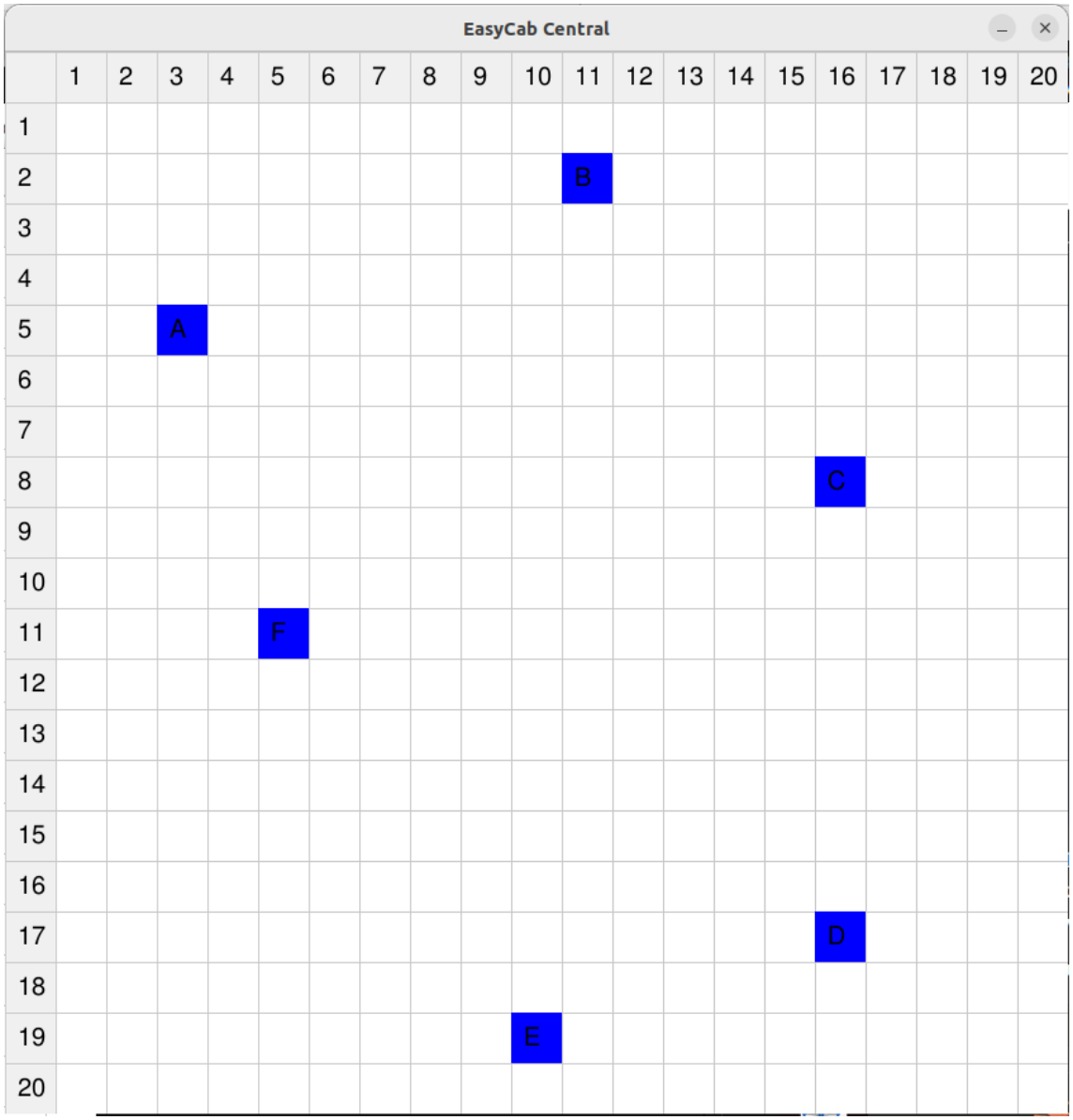
La interfaz gráfica, en consola o por un tablero, podemos usar pygame aquí también

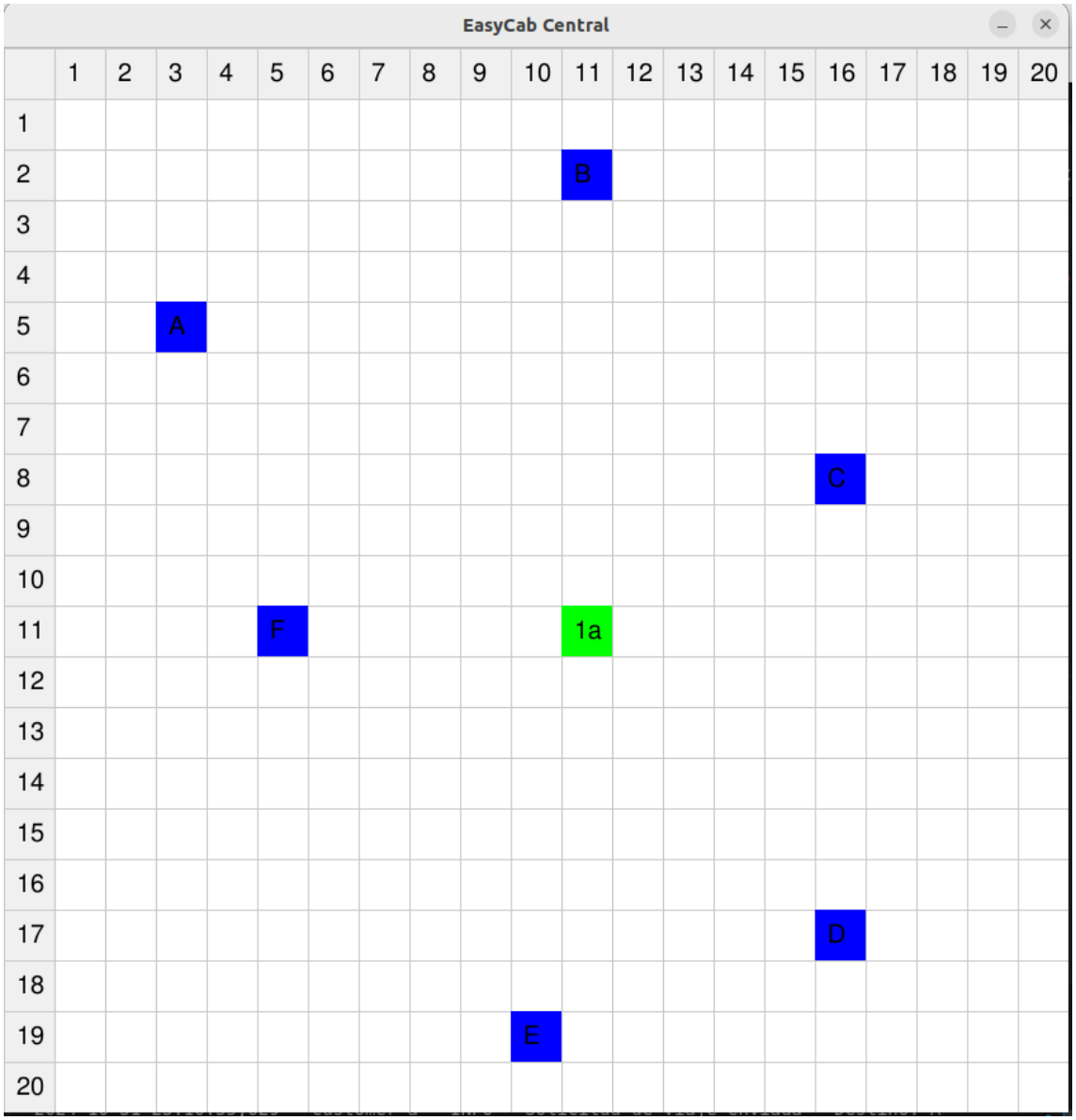
Para desarrollar esta interfaz gráfica en Python, puedes usar bibliotecas como:

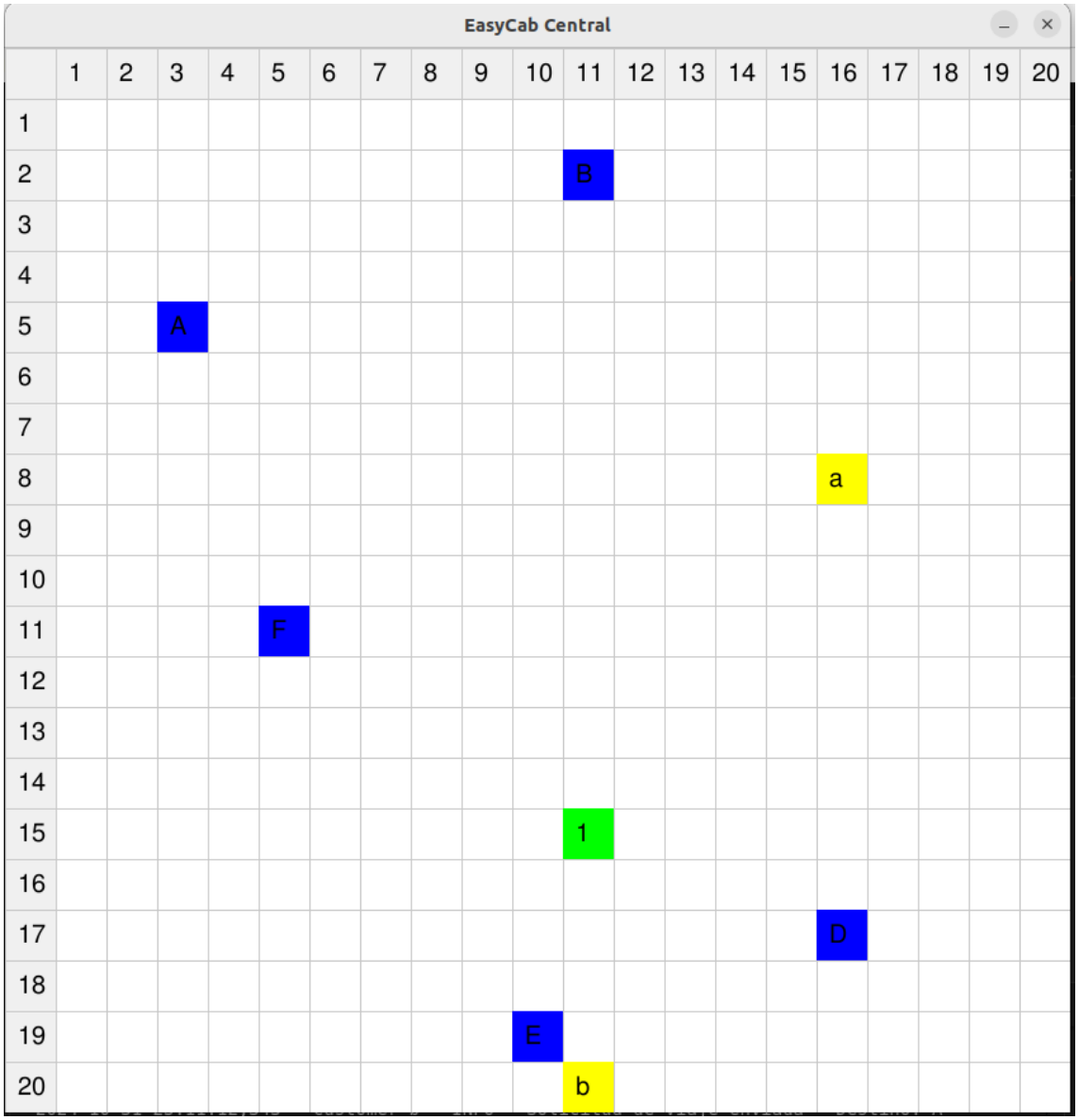
1. **Tkinter**: Es la biblioteca estándar de Python para interfaces gráficas. Puedes usarla para dibujar el tablero y representar los taxis y clientes.
   - Es simple y está bien integrada en Python.
   - Te permite dibujar objetos gráficos en un lienzo (canvas) y actualizar las posiciones de los taxis en tiempo real.
2. **Pygame**: Es una biblioteca más avanzada diseñada para crear juegos. Es perfecta si quieres más control gráfico y animaciones.
   - Puedes crear el tablero y mover los taxis con animaciones fluidas.
3. **Matplotlib**: Si prefieres gráficos más sencillos y visualizaciones más estáticas, esta biblioteca te permite dibujar un tablero y actualizar las posiciones, aunque es menos interactiva que Tkinter o Pygame.

FUNCIONAMIENTO:

MAPA INICIO




EasyCab Central
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
B
A
C
F
1a
D
E


EasyCab Central
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
B
A
a
F
1
D
E
b