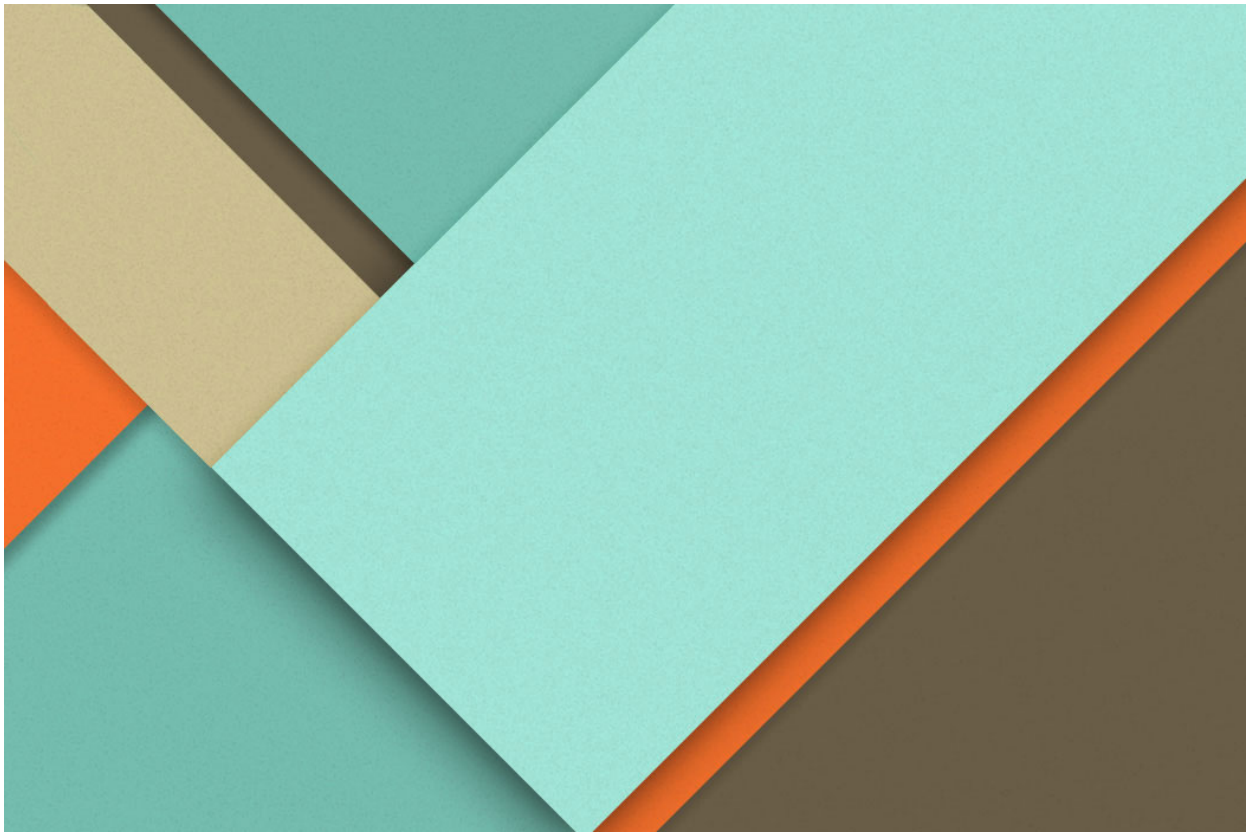


Practica 1 Sistemas Distribuidos

04/09/2023

—

Pablo García Valera

Jorge Ros Gómez

ÍNDICE

1. AD_DRON

Empezamos importando módulos y librerías de Python necesarios para su funcionamiento:

- **socket**: Proporciona funcionalidades para la comunicación por sockets (TCP/IP).
- **sys**: Proporciona acceso a algunas variables utilizadas o mantenidas por el intérprete y a funciones que interactúan con el intérprete.
- **json**: Permite trabajar con formato de datos JSON.
- **time**: Proporciona diversas funciones relacionadas con el tiempo.
- **KafkaConsumer** y **KafkaProducer** de kafka: Estos módulos permiten interactuar con un servidor Apache Kafka.

```
1   import socket
2   import sys
3   import json
4   import time
5   from kafka import KafkaConsumer,KafkaProducer
```

·Definimos una variable **file_Dron** con el nombre de un archivo JSON: 'Dron.json'

```
file_Dron= 'Dron.json'
```

·Definimos dos diccionarios **config_destinos** y **config_mapa** que contienen configuraciones para consumidores de Kafka. Estos diccionarios los utilizaremos para inicializar consumidores de Kafka más adelante en el código.

```
config_destinos = {
    'auto_offset_reset': 'earliest',
    'enable_auto_commit': True,
    'value_deserializer': lambda m: json.loads(m.decode('utf-8'))
}
config_mapa={
    'auto_offset_reset': 'lastest',
    'enable_auto_commit': False,
```

·Definimos varias funciones:

- **inicializar_productor(broker_address):** Inicializa un productor de Kafka y devuelve una instancia.
- **inicializar_consumidor(topic_name, broker_address, consumer_config=None):** Inicializa un consumidor de Kafka y devuelve una instancia. Puede aceptar una configuración personalizada opcional.
- **calcular_lrc(mensaje):** Calcula un valor de comprobación de redundancia longitudinal (LRC) para un mensaje dado.
- **incluir_json(file_Dron, dato):** donde leemos un archivo JSON, agrega un dato y luego lo vuelve a escribir en el archivo que es donde se almacena la información de todos los drones.

```python
def inicializar_productor(broker_address):
    return KafkaProducer(bootstrap_servers=broker_address, value_serializer=lambda v: json.dumps(v).encode('utf-8'))

def inicializar_consumidor(topic_name, broker_address, consumer_config=None):
    if consumer_config is None:
        consumer_config = {}

    # Configuracion
    default_config = {
        'bootstrap_servers': broker_address
    }

    final_config = {**default_config, **consumer_config}

    # Inicializar el consumidor con la configuración final
    consumer = KafkaConsumer(topic_name, **final_config)
    return consumer
def calcular_lrc(mensaje):
    bytes_mensaje = mensaje.encode('utf-8')

    lrc = 0
    # Calcular el LRC usando XOR
    for byte in bytes_mensaje:
        lrc ^= byte
     # Convertir el resultado a una cadena hexadecimal
    lrc_hex = format(lrc, '02X')
```

```python
def incluir_json(file_Dron, dato):

    try:

        with open(file_Dron, 'r') as file:
            try:
                json_data = json.load(file)
            except json.JSONDecodeError:
                    json_data = {}

        json_data.setdefault("lista_de_objetos", []).append(dato)

        with open(file_Dron, 'w') as file:
            json.dump(json_data, file, indent=2)  # indent para una escritura más bonita

    except FileNotFoundError:
        print(f'No se encontró el archivo en la ruta: {file_Dron}')

    except Exception as e:
        print(f'Ocurrió un error: {e}')
```

AD_Drone (la clase principal):

Atributos:

- **Alias**: Un alias o nombre asignado al dron.
- **id**: Un identificador único para el dron.
- **IP_Engine**: La dirección IP del motor (motor del dron).
- **Puerto_Engine**: El puerto de comunicación con el motor.
- **Ip_Puerto_Broker**: La dirección IP y puerto del servidor de Kafka.
- **IP_Registry**: La dirección IP del registro.
- **Puerto_Registry**: El puerto de registro.
- **posicion**: Una tupla que almacena la posición actual del dron (inicializada en (1, 1)).

```
class AD_Drone:
    #CREAMOS LA CLASE DRON
    def __init__(self,id,Alias,IP_Engine , Puerto_Engine, Ip_Puerto_Broker,IP_Registry , Puerto_Registry):
        self.Alias= Alias
        self.id =  id #id del dispositivo
        self.IP_Engine= IP_Engine
        self.Puerto_Engine= Puerto_Engine
        self.Ip_Puerto_Broker =  Ip_Puerto_Broker
        self.IP_Registry = IP_Registry
        self.Puerto_Registry= Puerto_Registry
        self.posicion = (1, 1)  # Posición inicial
```

Métodos:

- **conectar_al_servidor():** Intenta establecer una conexión con un servidor de registro. Envía un mensaje de solicitud de conexión <ENQ> y espera una respuesta <ACK>. Luego, muestra un menú para realizar acciones de registro.

```
def conectar_al_servidor(self):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cliente_conexion:
            servidor = (self.IP_Registry, self.Puerto_Registry)
            cliente_conexion.connect(servidor)
            enq = "<ENQ>"
            cliente_conexion.send(enq.encode())

            #Esperamos el ACK del servidor
            ack = cliente_conexion.recv(1024).decode()
            if ack == "<ACK>":
                print("Conexión exitosa.")
                opcion = input("option:\n1-Dar de alta\n2-Editar\n3-Dar de baja\n-->")
                self.ejecutar_menu_registrar(opcion,cliente_conexion)
            else:
                print(f"No hemos recibido el ACK, cerramos conexion: {ack}")
                cliente_conexion.close()

    except socket.error as err:
        print(f"Error de socket: {err}")
    except Exception as e:
        print(f"ERROR:  {e}")
```

- **unirse_espectaculo():** Intenta unirse a un espectáculo. Envia información del dron al motor y espera la autenticación <ACK>. Luego, escucha los mensajes de destino y mueve el dron hacia la posición especificada.

```
def unirse_espectaculo(self):

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as servidor:
        servidor.connect((self.IP_Engine, self.Puerto_Engine))

        data_token = f"<STX>{self.id},{self.token}<ETX>"
        data_token = data_token + calcular_lrc(data_token)
        servidor.send(data_token.encode())
        respuesta =  servidor.recv(1024).decode()
        if respuesta == "<ACK>":
            print("autenticacion correcta")
            consumer_destino =inicializar_consumidor('destinos',IP_Puerto_Broker,config_destinos)
            for mensajes in consumer_destino:
                # Procesa el primer mensaje recibido y luego rompe el bucle
                destinos =  mensajes.value
                break
            for drones in destinos["Drones"]:
                if drones["ID"]== self.id:
                    destino = tuple(map(int, drones["POS"].split(',')))
                    print(f"El dron con ID {self.id} debe moverse a la posición {destino}")
                    #logica para moverse
                    #self.posicion --> destino # cada vez que se mueva mandar un mensaje al topic movimiento

                    while self.posicion != destino:
                        self.mover_drone(destino)
                        #leer un topic de kafka que sea error
                        consumer_error= inicializar_consumidor('error_topic', IP_Puerto_Broker)
                        for mensajes in consumer_error:
                            mensaje_texto = mensajes.value.decode('utf-8')
                            print(f"Mensaje de error recibido: {mensaje_texto}")
                            destino= (1,1)
                            while self.posicion != destino:
                                self.mover_drone(destino)
                            print("Las condiciones son adversas volvemos a la base")
                            sys.exit(1)

                    #leer topic mapa y print
                    consumer_mapa = inicializar_consumidor('mapa', IP_Puerto_Broker,config_mapa)
                    for mensajes in consumer_mapa:
                        mapa = mensajes.value
                        print(mapa)
                    break

                else:
                    print(f"No se encontro destino para el dron con id:{self.id}")




        elif respuesta == "<NACK>":
            print("error de autenticaion")
        else:
            print("Error por identificar")
```

- **mover_drone(destino):** Mueve el dron hacia una posición de destino. Utiliza una lógica de movimiento para cambiar la posición actual del dron y publica los cambios de la posición actual en kafka.

```python
def mover_drone(self, destino):
    x_actual , y_actual = self.posicion
    x_final, y_final = map(int, destino['POS'].split(','))

    if x_actual < x_final:
        x_actual += 1
    elif x_actual > x_final:
        x_actual -= 1

    if y_actual < y_final:
        y_actual += 1
    elif y_actual > y_final:
        y_actual -= 1

    self.posicion = (x_actual,y_actual)


    time.sleep(1)  # Esperar un segundo entre cada movimiento
        # Publicar la ubicación en Kafka
    nuetro_topic = 'movimientos'
    payload = {
        'ID': self.id,
        'POS': f"{x_actual},{y_actual}"
    }
    producer.send(topic=nuetro_topic,value=payload)
    print(f"Dron {self.id} se mueve a {x_actual},{y_actual}")
```

- **registrar():** Inicia la lógica para registrar el dron en el servidor de registro.
- **ejecutar_menu_registrar(opcion, cliente_conexion):** Ejecuta el menú de registro en función de la opción seleccionada por el usuario.

```python
def registrar(self):
    #logica del registrar en AD_Registry

        try:
            self.conectar_al_servidor()
        except ConnectionRefusedError as e:
            print(f"Error de tipo: {e}")
        except (socket.error, OSError) as e:
            print(f"Error de socket: {e}")
        except Exception as e:
            print(f"Error: {e}")


def ejecutar_menu_registrar(self, opcion, cliente_conexion):
    try:
        if opcion == '1':
            self.Dar_alta(cliente_conexion)
        elif opcion == '2':
            self.Editar()
        elif opcion == '3':
            self.Dar_baja()
        else:
            print("Opción no válida.")
            sys.exit(1)
    except Exception as e:
        print(f"Error en la ejecución del menú: {e}")
        sys.exit(1)
```

- **Dar_alta(cliente_conexion):** Realiza la lógica para dar de alta el dron, envía información al servidor y recibe un token de autenticación.

```
def Dar_alta(self,cliente_conexion):

    stx, etx = "<STX>","<ETX>"
    dato =  {
        f"{self.id}":{
            "alias": f"{self.Alias}",
            "token": None
        }
    }


    json_dato= json.dumps(dato)
    lrc =  calcular_lrc(stx + json_dato + etx)
    envio=  stx+ json_dato +etx +lrc
    cliente_conexion.send(envio.encode())
    ack = cliente_conexion.recv(1024).decode()
    if ack == "<ACK>":
        print("Mensaje enviado correctamente")
        token = cliente_conexion.recv(1024).decode()
        dato[f"{id}"]['token'] =  token
        self.token = token
        incluir_json(file_Dron,dato)
        cliente_conexion.close()
```

- **separar_arg(arg):** Una función auxiliar que recibe un argumento en el formato IP:PUERTO y devuelve una tupla separada con la dirección IP y el puerto.

```
def separar_arg(arg):
    parte=arg.split(':')
    return parte[0] , int(parte[1])
```

·El bloque if __name__ == "__main__": se ejecuta si el script es ejecutado directamente, no si es importado como un módulo. En este bloque, se verifica si se proporcionan los argumentos necesarios al ejecutar el script. Si se proporcionan, se inicializan varios valores (como direcciones IP y puertos), se crea una instancia de la clase AD_Drone y se entra en un bucle de menú.

El menú ofrece tres opciones:

- Registrar un dron (drone.registrar()).
- Unirse a un espectáculo (drone.unirse_espectaculo()).
- Salir del programa (break).

```
if __name__ == "__main__":

    if len(sys.argv) !=  4:
        print("Error de argumentos")
        sys.exit(1)
    else:
        #registramos todos los puertos e ips introducidos por paramentros
        IP_Engine , Puerto_Engine =  separar_arg(sys.argv[1])
        IP_Puerto_Broker =  sys.argv[2]
        IP_Registry , Puerto_Registry =  separar_arg(sys.argv[3])
        producer =inicializar_productor(IP_Puerto_Broker)
        print("Puertos registrados...")
        id= int(input("Por favor, establece la ID del dispositivo\n-->"))
        Alias =  input("Por favor, establece el alias del dispositivo\n-->")
        # Crear una instancia de AD_Drone
        drone = AD_Drone(id,Alias, IP_Engine, Puerto_Engine, IP_Puerto_Broker, IP_Registry, Puerto_Registry)
        while True:
            menu = input("Elige una de las opciones:\n" +
                         "1-Registrar\n" +
                         "2-Unirse al espectaculo\n" +
                         "3-Salir\n-->")
            if menu == '1':
                drone.registrar()
            elif menu == '2':
                drone.unirse_espectaculo()
            elif menu == '3':
                print("Saliendo...")
                break
            else:
                print("Error de menú")
```

2. AD_ENGINE

·Empezamos importando módulos y definiendo unas constantes:

- Importamos varios módulos, como **time**, **sys**, **kafka**, **json**, **threading**, y **socket**, que se utilizan a lo largo del código.
- Se definen constantes para el formato de salida en la consola, como **GREEN**, **RED** y **END**.
- Se lee un archivo llamado "**bd_Engine.json**" para cargar datos relacionados con drones. Se trata de una especie de base de datos.

```
from time import sleep
import sys
from kafka import KafkaProducer , KafkaConsumer
from kafka.admin import KafkaAdminClient, NewTopic
from json import dumps
from threading import Lock
import json
import threading
import socket
import time



file_engine = 'bd_Engine.json'
condicion_drones = threading.Condition()
drones_autenticados = []
GREEN = '\033[92m'
RED = '\033[91m'
END = '\033[0m'
```

·Función **creacion_topics(administrador):**

- Define una función para crear topìcs Kafka que serán utilizados en el sistema. Los temas son "**destinos**", "**movimientos**", "**mapa**" y "**error_topic**".

```
def creacion_topics(administrador):
    #definimos los topics a crear
    topics = ["destinos","movimientos","mapa","error_topic"]
    num_partitions = 1
    replication_factor = 1
    #hacemos una lista de topics
    topic_nuevo =   [NewTopic(name=topic, num_partitions=num_partitions, replication_factor=replication_factor) for topic in topics]
    #los creamos mediante la instacia de administrador de kafka_admin
    administrador.create_topics(new_topics=topic_nuevo, validate_only=False)
```

·Función **verificar_drones_desconectados()**:

- Esta función se ejecuta en un hilo separado y verifica si algún dron ha perdido la conexión. Si un dron no ha enviado datos de movimiento durante 5 segundos, se elimina de la lista de drones autenticados.

```python
def verificar_drones_desconectados():
    while True:
        drones_a_eliminar = [drone for drone in drones_autenticados if drone.tiempo_sin_movimiento() > 5]
        for drone in drones_a_eliminar:
            print(f"El dron con id {drone.identificador} ha perdido la conexión y se ha eliminado")
            drones_autenticados.remove(drone)
        time.sleep(1)  # verifica cada segundo si los drones no se han movido en 5 segundos
```

·Función **consultar_clima(clima_servidor,flag, flag_lock):**

- También se ejecuta en un hilo separado y se comunica con un servidor de clima. Cada 5 segundos, consulta datos del clima para una ciudad (en este caso, "madrid") y realiza acciones en función de la temperatura. Si la temperatura es menor o igual a 0, establece una bandera para indicar que el clima es frío y cierra el programa porque no se puede hacer el espectáculo.

```python
def consultar_clima(clima_servidor,flag, flag_lock):
    while True:
        try:
            # Ciudad para consultar (reemplaza con la ciudad deseada)
            ciudad = "madrid"

            # Crear una solicitud en formato JSON
            solicitud = {"ciudad": ciudad}
            clima_servidor.send(json.dumps(solicitud).encode())

            # Recibir la respuesta del servidor de clima
            respuesta = clima_servidor.recv(1024).decode()
            datos_clima = json.loads(respuesta)
            temperatura = datos_clima["temperatura"]

            # Realizar acciones basadas en los datos del clima
            if float(temperatura) <= 0.0:
                with flag_lock:
                    flag[0] = True

        except Exception as e:
            print(f"Error al consultar el servidor de clima: {str(e)}")
            with flag_lock:
                flag[0] = True
            break

        time.sleep(5)  # Consulta cada 5 segundos
```

·Funciones para inicializar productores y consumidores Kafka:

- Estas funciones se utilizan para crear instancias de productores y consumidores Kafka que se conectan a un servidor Kafka.

```python
def inicializar_productor(broker_address):
    return KafkaProducer(bootstrap_servers=broker_address)


def inicializar_consumidor(topic, broker_address):
    consumer = KafkaConsumer(
        topic,
        bootstrap_servers=broker_address,
        value_deserializer=lambda x: json.loads(x.decode('utf-8'))
    )
    return consumer
```

·Función leer_destinos(file_destinos):

- Lee datos de destinos desde un archivo JSON llamado "**fichero_destinos.json**".

```python
def leer_destinos(file_destinos):


    with open(file_destinos, 'r') as file:
        try:
            json_data = json.load(file)
        except json.JSONDecodeError:

                json_data = {}
    return json_data
```

·Funciones para calcular LRC y desempaquetar datos:

- La función **calcular_lrc(mensaje)** calcula un valor LRC (Longitud Redundancia Cíclica) utilizando XOR para comprobar la integridad de los datos.

```python
def calcular_lrc(mensaje):
    bytes_mensaje = mensaje.encode('utf-8')

    lrc = 0
    # Calcular el LRC usando XOR
    for byte in bytes_mensaje:
        lrc ^= byte
    # Convertir el resultado a una cadena hexadecimal
    lrc_hex = format(lrc, '02X')

    return lrc_hex
```

- La función **desempaquetar_string(paquete)** verifica que los datos recibidos contienen marcadores <STX> y <ETX> y calcula su LRC para verificar la integridad de los datos y devuelve la DATA como una cadena.

```python
def desempaquetar_string(paquete):
    #COMPRUEBA QUE EXISTA EL STX
    inicio = paquete.find("<STX>")
    if inicio == -1:
        print("STX no encontrado")
        return None

    # COMPRUEBA QUE EXISTA EL ETX
    fin = paquete.find("<ETX>")
    if fin == -1:
        print("ETX no encontrado")
        return None

    data = paquete[inicio + len("<STX>"):fin]

    lrc_calculado = calcular_lrc(f"<STX>{data}<ETX>")

    # BUSCAMOS EL LRC DEL PAQUETE ORIGINAL
    lrc_inicio = fin + len("<ETX>")
    lrc_fin = lrc_inicio + 2
    lrc_paquete = paquete[lrc_inicio:lrc_fin]

    # Y LO COMPARAMOS
    if lrc_calculado != lrc_paquete:
        print(f"Error en LRC: {lrc_paquete} != {lrc_calculado}")
        return None

    # Devolver la DATA como una cadena
    return data
```

·Funciones para gestionar conexiones de drones:

- La función **escuchar_conexiones(servidor, ad_engine)** acepta conexiones entrantes de drones y las maneja en hilos separados.

```python
def escuchar_conexiones(servidor, ad_engine):
    while True:
        conexion, direccion = servidor.accept()
        print(f"Conexión entrante de {direccion}")
        threading.Thread(target=manejar_conexion, args=(conexion, ad_engine)).start()
```

- La función **iniciar_servidor(puerto, ad_engine)** crea un servidor de escucha en un puerto determinado.

```python
def iniciar_servidor(puerto, ad_engine):
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(("0.0.0.0", int(puerto)))
    servidor.listen(20)
    servidor.settimeout(90)

    print(f"Escuchando en el puerto {puerto}")
    threading.Thread(target=escuchar_conexiones, args=(servidor, ad_engine)).start()
```

·Función **manejar_conexion(conexión, ad_engine):**

- La función recibe la información del dron, verifica su autenticación utilizando el objeto **ad_engine**, crea una instancia de la clase Dron y la agrega a la lista **drones_autenticados** si la autenticación es exitosa. Luego, envía una confirmación de conexión o una negación al dron y cierra la conexión.

```
def manejar_conexion(conexion,ad_engine):
    #recibir datos del cliente (en este caso, asumimos que recibes id y token como cadena)
    global drones_autenticados, condicion_drones
    try:
        data = conexion.recv(1024).decode()
        data =desempaquetar_string(data)

        if not data:
            conexion.send("Formato incorrecto").encode()
            conexion.close()#le cerramos la conexion

        id_dron , token_dron = data.split(',')
        if ad_engine.conectar_dron(int(id_dron), token_dron):
            print( "Conexión exitosa")
            dron = Dron(identificador=int(id_dron),posicion=(0,0))
            with condicion_drones:
                drones_autenticados.append(dron)
                condicion_drones.notify()
            conexion.send("<ACK>".encode())
        else:
            print("No se pudo conectar el dron")
            conexion.send("<NACK>".encode())
    except Exception as e:
        print(f"Error: {e}")
        # Enviar respuesta al cliente
        conexion.send("<ACK>".encode())
    finally:
        # Cerrar la conexion
        conexion.close()
```

Función **pintar_mapa(drones_autenticados, dimension=20):**

Esta función recibe dos argumentos: drones_autenticados, y dimension. La función realiza lo siguiente:

- Convierte la lista de drones en un diccionario drones_dict donde la clave es el identificador del dron.
- Crea una matriz bidimensional llamada mapa con dimensiones dimension x dimension llena de espacios en blanco.
- Itera a través de los drones y actualiza el mapa con la posición de cada dron. Si varios drones están en la misma posición, se muestra solo uno de ellos (el de menor ID).
- Formatea el mapa en una representación de cadena con las posiciones de los drones.
- Muestra el mapa en la consola, donde los drones que llegaron a su destino se muestran en verde y los que no en rojo.
- Devuelve el mapa en forma de cadena.

```python
def pintar_mapa(drones_autenticados, dimension=20):


    # Convertir la lista de drones en un diccionario basado en el identificador del dron
    drones_dict = {dron.identificador: dron for dron in drones_autenticados}

    # Crear un mapa inicial vacío
    mapa = [[' ' for _ in range(dimension)] for _ in range(dimension)]

    for dron in drones_dict.values():
        x, y = dron.posicion
        if mapa[x][y] == ' ':
            mapa[x][y] = str(dron.identificador)
        else:
            # Si hay más de un dron en la misma posición, mostrar el dron con ID menor
            mapa[x][y] = str(min(int(mapa[x][y]), dron.identificador))

    linea = []
    linea.append('    ' + ' '.join([f"{i:02}" for i in range(dimension)]))
    for i in range(dimension):
        linea_elemento = []
        for cell in mapa[i]:
            if cell != ' ':
                dron_id = int(cell)
                if drones_dict[dron_id].llego_a_destino:
                    linea_elemento.append(GREEN + cell + END)
                else:
                    linea_elemento.append(RED + cell + END)
            else:
                linea_elemento.append(cell)
        linea.append(f"{i:02} | [" + "] [".join(linea_elemento) + "] | " + f"{i:02}")
    linea.append('    ' + ' '.join([f"{i:02}" for i in range(dimension)]))

    mapa = '\n'.join(linea)
    print(mapa)
    return mapa
```

·Clase **Dron:**

Atributos:

- **identificador**: Un identificador único para el dron.
- **posicion**: La posición actual del dron representada como una lista [x, y].
- **llego_a_destino**: Un indicador de si el dron llegó a su destino.
- **ultimo_movimiento**: El registro del tiempo del último movimiento del dron.

```python
def __init__(self, identificador, posicion):
    self.identificador = identificador
    self.posicion = list(posicion)
    self.llego_a_destino = False
    self.ultimo_movimiento = 0.0
```

Métodos**:**

- **llego_destino(self):** Marca el dron como que ha llegado a su destino.

```
def llego_destino(self):
    self.llego_a_destino = True
```

- **reset(self):** Restablece el dron a su estado inicial, marcándolo como no llegado a su destino y configurando su posición en [0, 0].

```
def reset(self):
    self.llego_a_destino =False
    self.posicion= [0,0]#posible fallo
```

- **actualizar_posicion(self, posicion):** Actualiza la posición del dron y registra el tiempo del último movimiento.

```
def actualizar_posicion(self, posicion):
    self.posicion = list(map(int, posicion.split(',')))
    self.ultimo_movimiento = time.time()
```

- t**iempo_sin_movimiento(self):** Calcula el tiempo transcurrido desde el último movimiento del dron en segundos.

```
def tiempo_sin_movimiento(self):
    if self.ultimo_movimiento is None:
        # Manejar adecuadamente si ultimo_movimiento es None
        # Por ejemplo, puedes inicializarlo con el tiempo actual
        self.ultimo_movimiento = time.time()
        return 0
    else:
        return time.time() - self.ultimo_movimiento
```

·Clase **Engine,** se encarga de gestionar la autenticación y la conexión de drones en el sistema**:**

- El constructor **__init__** se utiliza para cargar datos relacionados con los drones y su autenticación desde un archivo JSON llamado "**bd_Engine.json**". Inicializa el número de drones conectados en cero.

```
def __init__(self):
    with open(file_engine) as f:
        self.lista_de_objetos = json.load(f)['lista_de_objetos']

    self.drones_conectados = 0
```

- El método **verificar(self, id, token)** verifica si un dron con un ID y token específicos es auténtico. Itera a través de la lista de objetos cargados y compara los valores proporcionados con los datos de autenticación.

```
def verificar(self,id,token):
    print("entra verificar")
    with open(file_engine) as f:
        self.lista_de_objetos = json.load(f)['lista_de_objetos']
    print(self.lista_de_objetos)
    for dron_info in self.lista_de_objetos:
        if str(id) in dron_info and dron_info[str(id)]['token'] == token:

            return True
    return False
```

- El método **conectar_dron(self, id, token)** intenta conectar un dron verificando primero que no se haya alcanzado el límite de drones conectados (**drones_conectados** es menor que el número máximo de drones permitidos). Luego, verifica la autenticación del dron utilizando el método verificar. Si ambos criterios se cumplen, se permite la conexión y se incrementa el contador de drones conectados.

```
def conectar_dron(self, id, token):
    print(self.drones_conectados)

    if self.drones_conectados < int(numero_drones) and self.verificar(id, token):
        self.drones_conectados += 1
        return True
    else:
        print("false conectar")
        return False
```

·Bloque principal del programa:

- Procesa argumentos de línea de comandos, como el puerto de escucha, el número de drones, la dirección del broker Kafka y la dirección del servidor de clima.
- Crea temas Kafka necesarios, como "destinos", "movimientos", "mapa" y "error_topic".
- Lee datos de destinos desde un archivo JSON.
- Espera a que se conecten todos los drones necesarios antes de iniciar el espectáculo.
- Inicia hilos para verificar drones desconectados y consultar el servidor de clima.
- Inicia un bucle que forma figuras y actualiza la posición de los drones en función de los datos recibidos. Muestra el progreso en un mapa en la consola.
- Al finalizar el espectáculo, elimina los temas Kafka y cierra las conexiones.

```python
if __name__ == "__main__":



    if len(sys.argv) != 5:
        print("Error de argumentos..")
        sys.exit(1)



    contador_conexiones = 0
    file_destinos = "fichero_destinos.json"
    flag = [False]
    flag_lock =  Lock()
    motor = AD_Engine()
    puerto_escucha, numero_drones, ip_puerto_broker, ip_puerto_weather = sys.argv[1:5]
    ip_weather, puerto_weather = separar_arg(ip_puerto_weather)
    administrador  = KafkaAdminClient(bootstrap_servers = ip_puerto_broker)
    try:
        creacion_topics(administrador)
    except Exception as e:
        print(f"No se han creado los topics porque ya existen")

    destinos = leer_destinos(file_destinos)
    producer = inicializar_productor(ip_puerto_broker)
    consumer = inicializar_consumidor('movimiento', ip_puerto_broker)
    iniciar_servidor(puerto_escucha, motor)
    numero_drones_figura = len(destinos["figuras"][0]["Drones"])
    #esperamos que los drones necesarios se conecten desde el hilo
    with condicion_drones:
        while len(drones_autenticados) < numero_drones_figura:
            print("Esperando la conexion de drones necesarios para iniciar el espectaculo....")
            condicion_drones.wait()
    input("Se han conectado los drones necesarios, pulse cualquier tecla para iniciar el espectaculo")
    threading.Thread(target=verificar_drones_desconectados).start()

    #conectarse al servidor de clima
    clima_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clima_socket.connect((ip_weather, puerto_weather))
    threading.Thread(target=consultar_clima, args=(clima_socket,flag,flag_lock)).start()#por implementar

for figura in destinos["figuras"]:
    # Informa el inicio de la figura
    print(f"Comenzando a formar la figura: {figura['Nombre']}")

    # Manda la figura entera al broker destino
    producer.send('destinos', json.dumps(figura).encode('utf-8'))

    # hasta que todos los drones no llegen a su destino de figura ...
    while not all(drone.llego_a_destino for drone in drones_autenticados):
        with flag_lock:
            flag_actual = flag[0]
        if flag_actual:
            mensaje_error = "Error."
            producer.send('error_topic', value=mensaje_error.encode('utf-8'))
            producer.close()
            break

        for posicion_actualizada in consumer:
            for drone in drones_autenticados:
                if drone.identificador == posicion_actualizada["ID"] :
                        drone.actualizar_posicion(posicion_actualizada['POS'])
                        mapa = pintar_mapa(drones_autenticados)
                        producer.send('mapa', value=mapa.encode('utf-8'))
                        break
            if all(drone.llego_a_destino for drone in drones_autenticados):
                break

    # final de fig
    print(f"Figura {figura['Nombre']} completada")
    #Estaran con llega_A_destino en true
    for dron in drones_autenticados:
        dron.reset()
print("ha llegado el espectaculo al final")
topic_borrar = ["destinos", "movimientos","mapa"]
try:
    administrador.delete_topics(topic_borrar)
    producer.close()
    consumer.close()
    clima_socket.close()
except Exception as e:
    print(f"Error : {e}")
```

3. AD_REGISTRY

Empezamos importando módulos y definiendo variables:

- Importamos varios módulos necesarios, como **json**, **socket**, **sys**, **time**, y **threading**.
- Definimos el nombre del archivo donde se almacenará la información de los drones como **file_bd_engine**.

```
import json
import socket
import sys
import time
import threading
import string
import secrets

file_bd_engine =  'bd_Engine.json'
```

·Definimos varias funciones:

- **calcular_lrc(mensaje):** Calcula un valor de redundancia cíclica largo (LRC) para un mensaje dado. Este valor se utiliza para verificar la integridad de los paquetes recibidos.

```
def calcular_lrc(mensaje):
    bytes_mensaje = mensaje.encode('utf-8')

    lrc = 0
    # Calcular el LRC usando XOR
    for byte in bytes_mensaje:
        lrc ^= byte
    # Convertir el resultado a una cadena hexadecimal
    lrc_hex = format(lrc, '02X')

    return lrc_hex
```

- **incluir_json(file_Dron, dato):** Agrega información de un dron al archivo JSON especificado.

```
def incluir_json(file_Dron, dato):

    try:

        with open(file_Dron, 'r') as file:
            try:
                json_data = json.load(file)
            except json.JSONDecodeError:
                    json_data = {}

        json_data.setdefault("lista_de_objetos", []).append(dato)

        with open(file_Dron, 'w') as file:
            json.dump(json_data, file, indent=2)  # indent para una escritura más bonita

    except FileNotFoundError:
        print(f'No se encontró el archivo en la ruta: {file_Dron}')

    except Exception as e:
        print(f'Ocurrió un error: {e}')
```

- **genera_token():** Genera un token aleatorio de 7 caracteres para cada dron. Este token se utiliza como parte de la autenticación.

```python
def genera_token():

    caracteres = string.ascii_letters + string.digits
    token = ''.join(secrets.choice(caracteres) for _ in range(7))
    return token
```

- **desencriptar_paquete(paquete):** Desencripta un paquete que contiene un mensaje JSON y verifica su integridad utilizando un valor LRC. Si el paquete es válido, se extrae el mensaje JSON.

```python
def desencriptar_paquete(paquete):
    #COMPRUEBA QUE EXISTA EL STX
    inicio = paquete.find("<STX>")
    if inicio == -1:
        print("STX no encontrado")
        return None

    # COMPRUEBA QUE EXISTA EL ETX
    fin = paquete.find("<ETX>")
    if fin == -1:
        print("ETX no encontrado")
        return None

    #EXTRAE EL PAQUETE ELIMINANDO LAS CABECERAS
    data = paquete[inicio + len("<STX>"):fin]

    #REARAMOS EL PAQUETE ORIGINAL SIN EL LRC PARA CALCULARLO
    lrc_calculado = calcular_lrc(f"<STX>{data}<ETX>")

    # BUSCAMOS EL LRC DEL PAQUETE ORIGINAL
    lrc_inicio = fin + len("<ETX>")
    lrc_fin = lrc_inicio + 2
    lrc_paquete = paquete[lrc_inicio:lrc_fin]

    # Y LO COMPARAMOS
    if lrc_calculado != lrc_paquete:
        print(f"Error en LRC: {lrc_paquete} != {lrc_calculado}")
        return None

    # Decodificar la DATA (asumiendo que está en formato JSON)
    try:
        datos_json = json.loads(data)
        return datos_json
    except json.JSONDecodeError as e:
        print(f"Error al decodificar JSON: {e}")
        return None
```

·Función **procesar_cliente(cliente_conexion):**

- Esta función maneja la conexión con un cliente (dron) que envía un paquete encriptado.
- Comienza esperando un mensaje de solicitud <**ENQ**> del cliente. Si se recibe correctamente, se responde con un mensaje de confirmación <**ACK**>.
- A continuación, se recibe un paquete encriptado que se desencripta utilizando la función **desencriptar_paquete**. El paquete debe contener un mensaje JSON que incluye información sobre el dron, como su ID, posición, etc.
- Si el paquete es válido, se genera un token aleatorio con **genera_token(),** se asocia con la información del dron y se almacena en un archivo JSON utilizando la función **incluir_json**. El token se envía al dron para futuras autenticaciones.
- En caso de cualquier error, se cierra la conexión con el cliente.

```python
def procesar_cliente(cliente_conexion):
    try:

        enq = cliente_conexion.recv(1024).decode()

        if enq == "<ENQ>":
            ack = "<ACK>"
            cliente_conexion.send(ack.encode())
            mensaje = cliente_conexion.recv(1024).decode()

            mensaje = desencriptar_paquete(mensaje) #mensaje filtrado
            if mensaje is not None:


                cliente_conexion.send(ack.encode())
                token= genera_token()
                cliente_conexion.send(token.encode())

                try:

                    primera_clave = next(iter(mensaje))
                 # Cambiar el atributo "token" para la primera clave
                    mensaje[primera_clave]["token"] = token
                except StopIteration:
                    print("El objeto JSON está vacío")
                except Exception as e:
                    print(f'Ocurrió un error: {e}')

                incluir_json(file_bd_engine,mensaje)

        else:
            print("No llegó el <ENQ>")
            cliente_conexion.close()
    except socket.error as err:
        print(f"Error de socket: {err}")
    except Exception as err:
        print(f"Error de cliente: {err}")
    finally:
        cliente_conexion.close()
```

·Función **registro(puerto):**

- Inicia un servidor en el puerto especificado.
- Escucha las conexiones entrantes de los drones.
- Cuando un dron se conecta, se inicia un hilo para procesar su solicitud utilizando la función **procesar_cliente**.

```
def registro(puerto):
    try:
        #abrimos la conexion con sockets y nos ponemos a la escucha
        hostname = socket.gethostname()
        IPAddr = socket.gethostbyname(hostname)
        print("Your Computer IP Address is:" + IPAddr)
        conexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        conexion.bind(("0.0.0.0", puerto))
        conexion.listen(50)
        #establecemos un tiempo de maximo en el que el servidor no tiene conxiones y si no las tiene lo cierra
        conexion.settimeout(600)


        print(f"Escuchando en el puerto {puerto}...")
        while True:
            cliente_conexion, cliente_direccion = conexion.accept()
            print(f"Se ha establecido conexion con {cliente_direccion}")

            hilo = threading.Thread(target=procesar_cliente , args=(cliente_conexion,))
            hilo.start()

    except KeyboardInterrupt:
        print("Registro de Drones detenido.")
        conexion.close()
    except Exception as e:
        print(f"Error: {e}")
```

·El bloque if __name__ == "__main__" realiza las siguientes acciones:

- Comprueba si se proporciona el número correcto de argumentos en la línea de comandos. Se espera que se proporcione un solo argumento, que es el puerto en el que se ejecutará el servidor de registro de drones.
- Obtiene el valor del puerto desde los argumentos de línea de comandos (sys.argv[1]).
- Verifica si el valor del puerto es numérico utilizando el método isnumeric(). Si el puerto es un valor numérico válido, se llama a la función registro con el puerto convertido a un entero.
- Si el puerto no es un valor numérico válido, muestra un mensaje de error y finaliza el programa con el código de salida 1.

```
if __name__ == "__main__":
    if len(sys.argv) != 2:
            print("Error debes pasar los argumentos indicados (Puerto)")
            sys.exit(1)

    puerto = sys.argv[1]
    if puerto.isnumeric():  # Corregir aquí
        registro(int(puerto))
    else:
        print("Error de puerto")
        sys.exit(1)
```

4. AD_WEATHER

Empezamos importando módulos y variables:

- Importamos los módulos **sys**, **socket**, y **json** para realizar diversas operaciones.
- La variable ruta: Es una cadena que almacena la ruta de un archivo JSON llamado "**bd_Clima.json**". Este archivo se utiliza para almacenar datos de temperatura por ciudad.

```
import sys
import socket
import json


ruta = "bd_Clima.json"
```

·Función **ejecutar_servidor(puerto):**

- Esta función se encarga de iniciar un servidor en el puerto especificado.
- Crea un socket de tipo **SOCK_STREAM** para aceptar conexiones TCP entrantes.
- Escucha en el puerto especificado y espera una conexión.
- Cuando se establece una conexión, recibe datos del cliente (en este caso, se espera un JSON que contiene el nombre de la ciudad).
- Luego, carga datos de temperatura almacenados en el archivo JSON utilizando la función **cargar_datos_clima**().
- Busca la temperatura de la ciudad especificada en los datos cargados y envía una respuesta JSON que contiene el nombre de la ciudad y su temperatura al cliente.
- El ciclo se repite para atender múltiples solicitudes de clientes.
- Si se produce un error, se captura y se muestra un mensaje de error. Finalmente, se cierra la conexión y el servidor.

```python
def ejecutar_servidor(puerto):

    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(("localhost", int(puerto)))
    servidor.listen(1)
    print(f"Esperando conexión en el puerto {puerto}...")

    # Aceptar una nueva conexión
    conn, addr = servidor.accept()
    print(f"Conexión establecida desde {addr}")

    try:
        while True:
            data = conn.recv(1024).decode()
            if not data:
                break  # Si no hay datos, el cliente se desconectó

            ciudad = json.loads(data)["ciudad"]
            datos = cargar_datos_clima()

            temperatura = datos.get(ciudad, {"temperatura": 15})["temperatura"]
            respuesta = {"ciudad": ciudad, "temperatura": temperatura}

            conn.send(json.dumps(respuesta).encode())

    except Exception as e:
        print(f"Error: {e}")
    finally:
        conn.close()
        servidor.close()
```

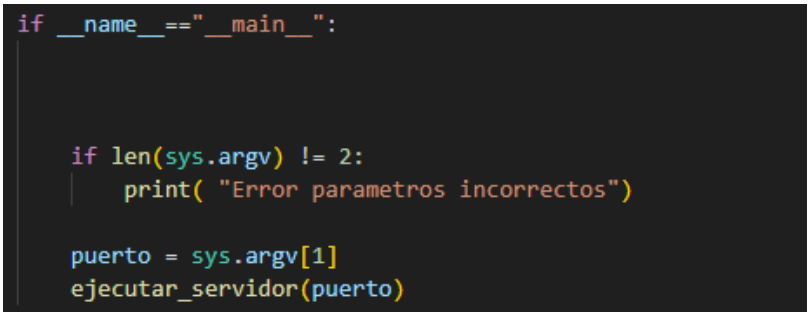
·Función **cargar_datos_clima():**

- Esta función se utiliza para cargar los datos de temperatura almacenados en el archivo JSON "**bd_Clima.json**".
- Abre el archivo en modo lectura y carga su contenido en un diccionario utilizando json.load().
- Si el archivo JSON está vacío o tiene un formato incorrecto, se asigna un diccionario vacío al objeto datos.
- La función devuelve el diccionario datos.

```python
def cargar_datos_clima():
    with open(ruta, "r") as file:
        try:
            datos = json.load(file)
        except json.JSONDecodeError:
            datos = {}
            print("Error: El archivo JSON está vacío o tiene un formato incorrecto.")
    return datos
```

·Bloque Principal if __name__=="__main__":

- En el bloque principal, se verifica si se proporciona el número correcto de argumentos en la línea de comandos. Debe proporcionarse exactamente un argumento, que es el puerto en el que se ejecutará el servidor.
- Si se proporciona el puerto, se llama a la función **ejecutar_servidor(puerto)** para iniciar el servidor y escuchar en el puerto especificado.



5. ARCHIVOS JSON

BD_Engine.json

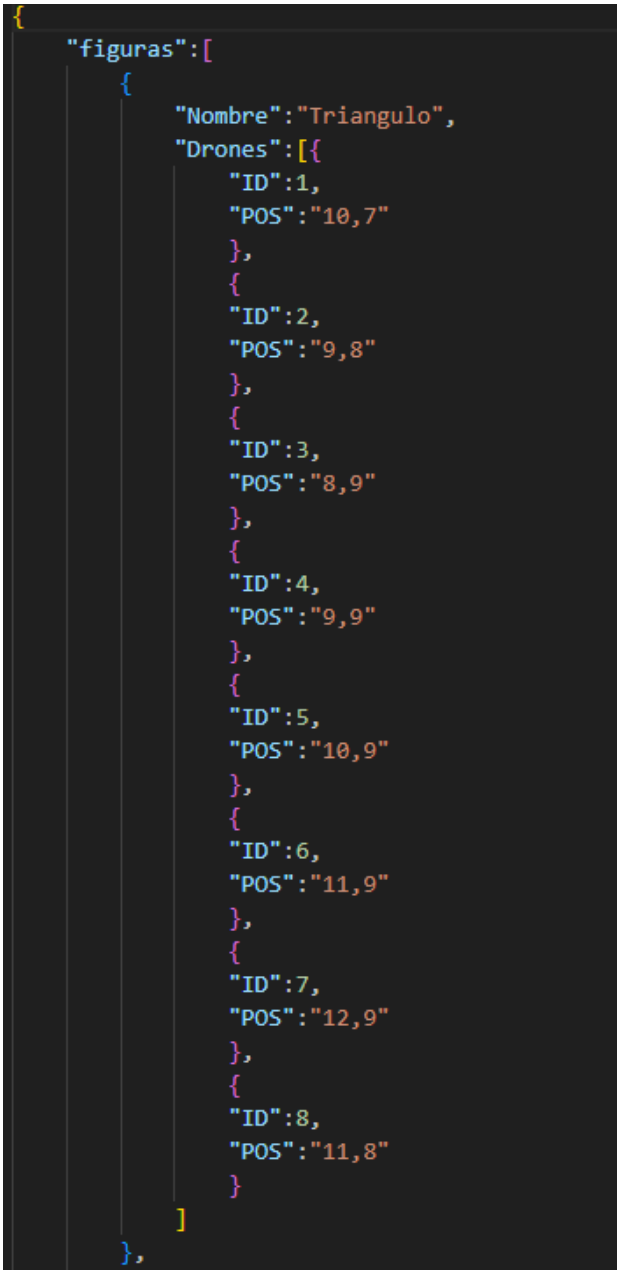

```
{
  "lista_de_objetos": [
    {
      "1": {
        "alias": "jorge",
        "token": "fIoEmBV"
      }
    },
    {
      "2": {
        "alias": "a",
        "token": "ue68m56"
      }
    },
    {
      "3": {
        "alias": "a",
        "token": "1n2cMA9"
      }
    },
    {
      "4": {
        "alias": "a",
        "token": "ImYSlih"
      }
    },
    {
      "5": {
        "alias": "trt",
        "token": "bUZi82H"
      }
    },
    {
      "6": {
        "alias": "43fds",
        "token": "CuJ0RvB"
      }
    },
    {
      "7": {
        "alias": "jorge",
        "token": "sLRJyaJ"
      }
    },
    {
      "8": {
        "alias": "vjosda",
        "token": "txEjqNp"
```

BD_Clima.json

```
{
"lista ciudades":
[{
    "ciudad": "madrid",
    "temperatura":"33.2"
},
{
    "ciudad": "sevilla",
    "temperatura":"40.3"
},
{
    "ciudad":"barcelona",
    "temperatura":"42.0"
}

]
}
```

Fichero_destinos.json



```
{
    "figuras":[
        {
            "Nombre":"Triangulo",
            "Drones":[{
                "ID":1,
                "POS":"10,7"
                },
                {
                "ID":2,
                "POS":"9,8"
                },
                {
                "ID":3,
                "POS":"8,9"
                },
                {
                "ID":4,
                "POS":"9,9"
                },
                {
                "ID":5,
                "POS":"10,9"
                },
                {
                "ID":6,
                "POS":"11,9"
                },
                {
                "ID":7,
                "POS":"12,9"
                },
                {
                "ID":8,
                "POS":"11,8"
                }
            ]
        },
```

Dron.json


```
{
  "lista_de_objetos": [
    {
      "1": {
        "alias": "jorge",
        "token": "RCTvFGk"
      }
    },
    {
      "1": {
        "alias": "JORGE ROS",
        "token": "kedJzL2"
      }
    },
    {
      "2": {
        "alias": "PP",
        "token": "yrtTLGK"
      }
    },
    {
      "3": {
        "alias": "JORGE",
        "token": "YzGDp9D"
      }
    },
    {
      "4": {
        "alias": "PEPE",
        "token": "u4fpZL8"
      }
    },
    {
      "5": {
        "alias": "D",
        "token": "fI3Ea5U"
      }
    },
    {
      "6": {
        "alias": "CDS",
        "token": "EtpJWr2"
      }
```