

5 DE NOVIEMBRE DE 2023

**SISTEMAS DISTRIBUIDOS**

# **SHOW DE DRONES**



**Victor Dominguez Rodriguez**

44939671V

# INFORME PRACT-1

**P**ara el desarrollo de la practica se uso IDE Visual Studio Code y lenguaje Python, kafka como principal herramienta de comunicación, json para guardar datos y sockets para comunicaciones básicas del registry y el weather.

## CLASE ENGINE

### Clase Mapa

He creado una clase `Mapa` en Python para gestionar un entorno bidimensional donde se pueden mover y visualizar drones. La clase funciona de la siguiente manera

1. Al iniciar una instancia de `Mapa`, establezco un tamaño predeterminado de 20x20 para la cuadrícula que representa el mapa. Relleno esta cuadrícula con espacios en blanco y delimito los bordes con puntos para distinguir claramente los límites del área de trabajo.
2. Mantengo un registro de las posiciones de los drones mediante un diccionario `drones\_positions`, que almacena sus coordenadas y colores. Además, he añadido un diccionario `dron\_solapado` para tratar la situación donde dos drones comparten la misma posición en el mapa.
3. He definido el método `update\_position` para mover un dron a una nueva posición. Este método comprueba primero si la nueva posición está dentro de los límites del mapa. Si un dron se mueve de una posición donde previamente había otro dron, me aseguro de volver a mostrar el dron que estaba oculto. Además, si la nueva posición resulta estar ocupada, actualizo el registro de posiciones solapadas.
4. Utilizo el método `place\_drone` para actualizar visualmente la posición del dron en el mapa. Aquí aplico colores usando la biblioteca `colorama` para diferenciar entre drones rojos y verdes, donde cada color tiene un código de fondo y de texto específico. Esto hace que la visualización sea más intuitiva y fácil de seguir.
5. Finalmente, tengo un método `display` que imprime la cuadrícula del mapa en la consola. Uso la conversión de elementos a cadenas para que la cuadrícula se imprima correctamente y sea legible para los usuarios.



Mi intención con esta clase es que sea una parte integral de un sistema más grande que pueda simular el movimiento de drones o actuar como interfaz para su operación en tiempo real, proporcionando retroalimentación visual y precisa en una terminal que soporte colores ANSI.

```
vdr@MacBook-Pro-de-Victor SD % cd Nucleo  
vdr@MacBook-Pro-de-Victor Nucleo % python3.9 Mapa.py  
vdr@MacBook-Pro-de-Victor Nucleo % python3.9 Mapa.py
```

	ID	X	Y
Triángulo	1	10	7
Triángulo	2	9	8

```
vdr@MacBook-Pro-de-Victor Nucleo %
```

## Clase Engine

He desarrollado una clase `Engine` en Python que actúa como un sistema de control centralizado para operar y gestionar drones dentro de un entorno de simulación. La clase se encarga de iniciar y reiniciar el mapa, autenticar drones, y coordinar la visualización y las acciones de los drones. Aquí está un desglose de lo que hago:

Inicialización: Al crear una instancia de `Engine`, inicializo un conjunto para rastrear drones confirmados, creo un nuevo `Mapa`, y establezco variables para la ciudad y el estado del clima.

Actualización de Mapa: Con ``update_map``, extraigo la información de un mensaje recibido, que incluye ID del dron, coordenadas y color, y actualizo la posición del dron en el mapa.

Visualización del Mapa: El método ``display_map`` me permite mostrar el mapa en ciclos regulares, proporcionando una representación visual en tiempo real de la ubicación de los drones.

Carga de Drones: ``load_drones`` se utiliza para cargar los datos de los drones desde un archivo, manejando excepciones en caso de que el archivo no exista.

Geometría del Mapa: Implemento ``wrap_coordinates`` para manejar la lógica de un mundo "envuelto", permitiendo que los drones que se muevan fuera de un borde reaparezcan en el opuesto, emulando una geometría esférica.

Manejo de Conexiones: En ``handle_connection``, gestiono las conexiones entrantes, autenticando drones basándome en los datos recibidos y enviando una respuesta adecuada.

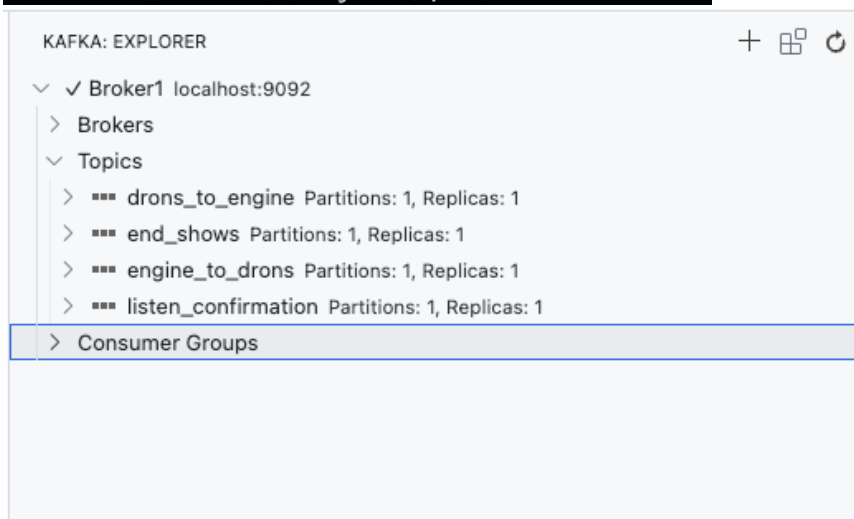
Autenticación: ``authenticate`` es un bucle que mantiene mi servidor escuchando por conexiones entrantes de drones y llamando al método para manejar cada conexión.

Kafka Producer para Shows: Con ``start_show``, me comunico con drones a través de Kafka, enviando coordenadas para formaciones de figuras predefinidas, manejando confirmaciones y condiciones climáticas para continuar o terminar el espectáculo de drones.

Confirmaciones de Drones: ``all_drones_confirmed`` verifica si todos los drones necesarios para una figura han confirmado su posición, lo cual es esencial para sincronizar el inicio de cada parte del espectáculo.

Escuchar Confirmaciones: Con ``listen_for_confirmations``, escucho confirmaciones de los drones a través de un consumidor Kafka para saber cuándo han alcanzado su posición destinada.

Vuelta a la Base: ``backToBase`` se encarga de manejar el procedimiento para que todos los drones regresen a su punto de partida al final del espectáculo.

[illegible]

## Iniciando la Escucha de Drones

Primero, inicio la escucha de los drones a través de Kafka, un sistema de mensajería distribuido. Configuro el consumidor para recibir mensajes del topic 'drons\_to\_engine' y empiezo a procesar cada mensaje a medida que llega.

### Carga y Guardado de Actualizaciones

Cargo las últimas actualizaciones de los drones desde un archivo JSON. Si el archivo no existe, comienzo con una lista vacía. También tengo una función para guardar el estado actual de todos los drones conocidos en este archivo. Esto asegura que mantengo un registro persistente del estado de cada dron.

### Procesamiento de Mensajes

Cuando recibo un mensaje de un dron, actualizo su estado y coordenadas en mi lista interna. Si un dron está aterrizando o moviéndose, actualizo el mapa correspondiente con su nueva ubicación.

### Interacción con el Servicio Climático

Realizo llamadas a un servidor externo para obtener información del clima para una ciudad específica. Esta información me permite tomar decisiones inteligentes sobre si el espectáculo debe continuar o detenerse en función de las condiciones climáticas. Envío alertas si las condiciones son desfavorables y tomo la decisión de terminar el show si es necesario.

### Chequeo Periódico del Clima

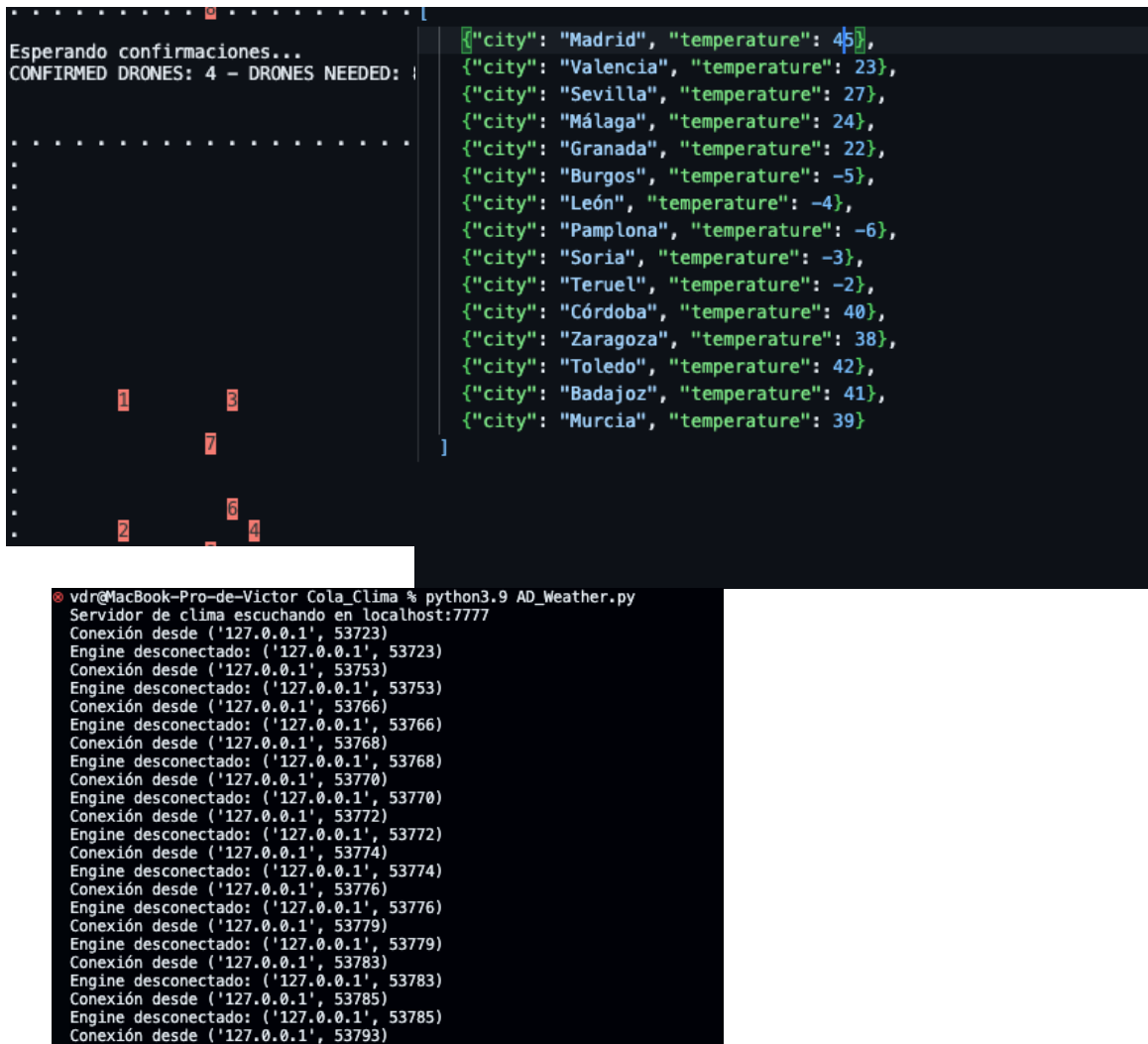
Tengo una función dedicada a verificar el clima cada 10 segundos. Esto me ayuda a actuar rápidamente en caso de un cambio inesperado en las condiciones climáticas que podría afectar el espectáculo de drones.

### Terminando el Espectáculo

Si las condiciones climáticas son malas o si recibo una instrucción para terminar el espectáculo, produzco un mensaje en el topic 'end\_show' de Kafka, informando a todos los drones y sistemas relacionados que el espectáculo debe concluir.

### Inicio del Motor

Lanzo varias tareas en hilos paralelos para manejar múltiples partes del espectáculo simultáneamente. Esto incluye iniciar el espectáculo, escuchar actualizaciones de drones, mostrar el mapa, confirmar el aterrizaje de drones, revisar el clima y finalizar el show. Utilizo **threading.Thread** para manejar la concurrencia en Python.



The screenshot displays a Python application interface. On the left, a terminal window shows the text "Esperando confirmaciones..." and "CONFIRMED DRONES: 4 - DRONES NEEDED:". Below this, a map area contains several red square markers, each labeled with a number (1, 2, 3, 4, 6, 7). On the right, a code editor shows a list of city temperature data in JSON format. Below the code editor, a terminal window shows the command "python3.9 AD\_Weather.py" and its output, which includes a series of connection and disconnection messages for various IP addresses and ports.

```
Esperando confirmaciones...
CONFIRMED DRONES: 4 - DRONES NEEDED:

[{"city": "Madrid", "temperature": 45},
{"city": "Valencia", "temperature": 23},
{"city": "Sevilla", "temperature": 27},
{"city": "Málaga", "temperature": 24},
{"city": "Granada", "temperature": 22},
{"city": "Burgos", "temperature": -5},
{"city": "León", "temperature": -4},
{"city": "Pamplona", "temperature": -6},
{"city": "Soria", "temperature": -3},
{"city": "Teruel", "temperature": -2},
{"city": "Córdoba", "temperature": 40},
{"city": "Zaragoza", "temperature": 38},
{"city": "Toledo", "temperature": 42},
{"city": "Badajoz", "temperature": 41},
{"city": "Murcia", "temperature": 39}]

vdr@MacBook-Pro-de-Victor Cola_Clima % python3.9 AD_Weather.py
Servidor de clima escuchando en localhost:7777
Conexión desde ('127.0.0.1', 53723)
Engine desconectado: ('127.0.0.1', 53723)
Conexión desde ('127.0.0.1', 53753)
Engine desconectado: ('127.0.0.1', 53753)
Conexión desde ('127.0.0.1', 53766)
Engine desconectado: ('127.0.0.1', 53766)
Conexión desde ('127.0.0.1', 53768)
Engine desconectado: ('127.0.0.1', 53768)
Conexión desde ('127.0.0.1', 53770)
Engine desconectado: ('127.0.0.1', 53770)
Conexión desde ('127.0.0.1', 53772)
Engine desconectado: ('127.0.0.1', 53772)
Conexión desde ('127.0.0.1', 53774)
Engine desconectado: ('127.0.0.1', 53774)
Conexión desde ('127.0.0.1', 53776)
Engine desconectado: ('127.0.0.1', 53776)
Conexión desde ('127.0.0.1', 53779)
Engine desconectado: ('127.0.0.1', 53779)
Conexión desde ('127.0.0.1', 53783)
Engine desconectado: ('127.0.0.1', 53783)
Conexión desde ('127.0.0.1', 53785)
Engine desconectado: ('127.0.0.1', 53785)
Conexión desde ('127.0.0.1', 53793)
```





```
Nucleo > {} last_updates.json > ...
4      "COORDENADA_X_ACTUAL": 10,
5      "COORDENADA_Y_ACTUAL": 12,
6      "ESTADO_ACTUAL": "MOVING"
7    },
8    {
9      "ID_DRON": 5,
10     "COORDENADA_X_ACTUAL": 8,
11     "COORDENADA_Y_ACTUAL": 8,
12     "ESTADO_ACTUAL": "MOVING"
13   },
14   {
15     "ID_DRON": 2,
16     "COORDENADA_X_ACTUAL": 9,
17     "COORDENADA_Y_ACTUAL": 10,
18     "ESTADO_ACTUAL": "LANDED"
19   },
20   {
21     "ID_DRON": 6,
22     "COORDENADA_X_ACTUAL": 10,
23     "COORDENADA_Y_ACTUAL": 17,
24     "ESTADO_ACTUAL": "MOVING"
25   },
26   {
27     "ID_DRON": 8,
28     "COORDENADA_X_ACTUAL": 9,
29     "COORDENADA_Y_ACTUAL": 19,
```

## CLASE REGISTRY

La clase `Registry` es un servicio de registro para drones, implementado en Python. A continuación se proporciona un resumen de sus características y funciones:

- Direcciones y puertos: La clase opera en `localhost` en el puerto `4444`. Usa estos parámetros para configurar su socket.
- Persistencia de datos: Maneja un archivo JSON (`'drones_registry.json'`) donde almacena y recupera información sobre drones registrados.
- Funciones de carga y guardado:
  - `load_drones()`: Carga la lista de drones registrados desde el archivo JSON.
  - `save_drones(drones)`: Guarda la lista actualizada de drones en el archivo JSON.

Registro y manejo de tokens:

- `register_dron(token)`: Acepta un token y lo registra si aún no existe en la lista de drones.
- `get_token_for_dron(dron_id)`: Genera y devuelve un token único para un nuevo dron, basado en su ID. Si el dron ya está registrado, devuelve su token actual.
- `non_register_dron(dron_id)`: Verifica si un dron no está registrado en la lista.

## Manejo de conexiones:

- `handle_connection(conn, addr)`: Maneja las conexiones entrantes. Recibe el ID del dron, verifica si ya está registrado y, si no, registra el dron y le asigna un token nuevo. Envía una respuesta al cliente con el resultado del registro.

## -Socket y escucha de conexiones:

- Se crea un socket utilizando el protocolo TCP/IP.
- Se vincula el socket al host y puerto especificados.
- Se configura el socket para que escuche conexiones entrantes.
- En un bucle infinito, acepta conexiones y para cada una, se invoca `handle_connection` para gestionar la solicitud correspondiente.

The screenshot displays a development environment with two main panels. The left panel shows the source code for `AD_Registry.py`, which includes functions for saving drones, getting tokens, and handling connections. The right panel shows the `drones_registry.json` file containing a list of registered drones with their IDs and tokens. Below these panels is a terminal window showing the execution of the program. The terminal output indicates that the registry is listening on port 4444 and that a drone with ID 52 has been successfully registered with a specific token.

```
Nucleo > AD_Registry.py > handle_connection
39     with open(FILENAME, 'r') as file:
40         return json.load(file)
41     except FileNotFoundError:
42         return []
43
44 def save_drones(drones):
45     with open(FILENAME, 'w') as file:
46         json.dump(drones, file)
47
48 def get_token_for_dron(dron_id):
49     drones = load_drones()
50
51     # Buscar si el dron ya está registrado
52     for dron in drones:
53         if dron["id"] == dron_id:
54             return dron["token"]
55
56     # Si no está registrado, generar un nuevo token único
57     existing_tokens = [drone["token"] for drone in drones]
58     new_token = str(uuid.uuid4())
59     while new_token in existing_tokens:
60         new_token = str(uuid.uuid4())
61
62     # Añadir el nuevo dron y su token al registro
63     drones.append({"id": dron_id, "token": new_token})
64     save_drones(drones)
65
66     return new_token
67
Nucleo > {} drones_registry.json > ...
1 [{"id": 2, "token": "148864d4-759c-41fa-8923-86a34b0029f0"},
2  {"id": 3, "token": "075db77a-2a6e-40bf-ada4-9398c63361a6"},
3  {"id": 4, "token": "888a7ab6-2781-4210-981f-c66e649621e8"},
4  {"id": 7, "token": "3c8e7281-0260-4672-a7b2-d1f0556ac889"}]

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
vdr@MacBook-Pro-de-Victor Dron % python3.9 AD_Dron.py
Selección un ID de dron entre 1 y 100:
52

Selección una opción:
1. Registrar el dron en el Registry
2. Autenticar el dron con el Engine
3. Comenzar a escuchar instrucciones
4. Salir
Selección una opción: 1
Dron 52 registrado exitosamente con el token: 32a4de04-36e5-4cdd-ac1a-d2165aece5ab

Selección una opción:
1. Registrar el dron en el Registry
2. Autenticar el dron con el Engine
3. Comenzar a escuchar instrucciones
4. Salir

vdr@MacBook-Pro-de-Victor Nucleo % python3.9 AD_Registry.py
Registry escuchando en 4444
Conectado por ('127.0.0.1', 55009)
Dron 32a4de04-36e5-4cdd-ac1a-d2165aece5ab registrado exitosamente.
```

## CLASE WEAHTER

La clase `AD_WEATHER` en el código proporcionado es una implementación de un servidor de clima que utiliza sockets para comunicarse. A continuación se describe su funcionalidad y estructura:

Inicialización:

- `__init__(self, weather_file, host, port)`: Constructor de la clase que inicializa la instancia con la ubicación del archivo de datos de clima (`weather_file`), el host y el puerto para el servidor socket.

Manejo de datos de clima:

- `load_weather_data(self)`: Carga los datos de clima desde un archivo JSON especificado. Si el archivo no se encuentra, imprime un mensaje de error y retorna `None`.

- `get_temperature(self, city)`: Obtiene la temperatura de una ciudad específica. Si los datos de la ciudad están disponibles, devuelve la temperatura; de lo contrario, devuelve "Unknown".

Funcionamiento del servidor:

- `start_server(self)`: Este método configura y ejecuta un servidor socket que escucha en el host y puerto especificados. Acepta conexiones entrantes y, para cada cliente, recibe un nombre de ciudad, busca la temperatura para esa ciudad y devuelve la información al cliente.

### Excepciones y errores:

- El servidor maneja varios tipos de excepciones: si ocurre un error al aceptar conexiones o comunicarse con un cliente, imprime un mensaje de error y continúa. Si hay un problema al decodificar los datos JSON, imprime un mensaje de error y rompe el ciclo de comunicación con el cliente actual.
- Si el servidor no puede iniciarse debido a un error de socket, imprime un mensaje de error y sale del programa con ``sys.exit(1)``.

### Función principal:

- ``main()``: Esta función crea una instancia de ``AD_WEATHER`` con un archivo de condiciones meteorológicas (``"weather_conditions.json"``), el host local y el puerto ``7777``. Luego, inicia el servidor llamando a ``start_server()``.

### Punto de entrada del programa:

- ``if __name__ == "__main__": main()``: Si el script se ejecuta como programa principal (no se importa desde otro módulo), se llama a la función ``main()``.

En resumen, la clase ``AD_WEATHER`` sirve como un servidor que proporciona datos de temperatura para ciudades solicitadas a través de un cliente de socket. La comunicación se basa en el protocolo TCP y se utiliza JSON para el intercambio de datos.

## 13



## CLASE DRON

Variables de configuración: Direcciones y puertos para el `Registry`, el `Engine` y el servidor de Kafka (`Broker`).

Variables de estado: Estados (`STATES`) y colores (`COLORS`) que puede tomar un dron.

Inicialización (`\_\_init\_\_`): Se establece la ID, token, estado, posición y color iniciales del dron.

Registro (`register`): Se conecta al `Registry` para obtener un token de autenticación.

Autenticación (`authenticate`): Usa el token para autenticarse con el `Engine`.

Recepción de datos (`recive\_data`): Suscribe al dron a un tópico Kafka para recibir instrucciones del `Engine`.

Procesamiento de mensajes (`process\_message`): Extrae instrucciones del mensaje de Kafka y llama a `run` para mover el dron.

Movimiento (`move\_one\_step`): Mueve el dron un paso hacia la posición objetivo.

Ejecución de la ruta (`run`): Lleva al dron hacia la posición objetivo y reporta su estado al `Engine`.

Finalización de la presentación (`endShow`): Espera por un mensaje de Kafka que indique que el espectáculo ha terminado.

Comunicación con el Engine:

Actualizar estado (`send\_update`): Envía el estado actual al `Engine`.

Confirmar posición (`send\_confirmation`): Confirma al `Engine` la llegada a la posición objetivo.

Ejecución del dron (`run\_dron`): Inicia hilos para procesos simultáneos como recibir datos y enviar actualizaciones.

Flujo del Programa:

-Menú (`menu`): Permite al usuario elegir entre registrar el dron, autenticarlo, comenzar a escuchar instrucciones o salir.

-Main (`main`): Punto de entrada del programa que llama al menú.

Funcionalidad Adicional:

- La clase implementa métodos para manejar las propiedades del dron con `getters` y `setters`.

- Contiene un método `destroy`, aunque no está implementado para realizar ninguna acción.

- Las interacciones con el `Registry` y el `Engine` se realizan mediante conexiones de socket, enviando y recibiendo mensajes en formato JSON.

- Las operaciones relacionadas con Kafka utilizan un consumidor para recibir datos y un productor para enviar mensajes a distintos tópicos.

- La lógica de movimiento implementa un simple algoritmo paso a paso hacia la posición objetivo.

- El método `run\_dron` utiliza multithreading para manejar varias tareas de forma concurrente.
- La función `main` esencialmente invoca al menú, y la ejecución del programa comienza desde el bloque `if \_\_name\_\_ == '\_\_main\_\_':`, llamando a `main()`.

```

Seleccione una opcion:
1. Registrar el dron en el Registry
2. Autenticar el dron con el Engine
3. Comenzar a escuchar instrucciones
4. Salir
Seleccione una opcion: 3
SEND_UPDATE ID de Dron: 7
SEND_CONFIRMATION: WAITING
P:(2, 2), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(3, 3), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(4, 4), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(5, 5), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(6, 6), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(7, 7), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(8, 8), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(9, 9), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(10, 10), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(11, 11), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(12, 12), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(13, 13), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(13, 14), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
P:(13, 15), S: MOVING, M: (13, 15)
SEND_UPDATE ID de Dron: 7
SEND_UPDATE ID de Dron: 7
SEND_CONFIRMATION: LANDED

```

