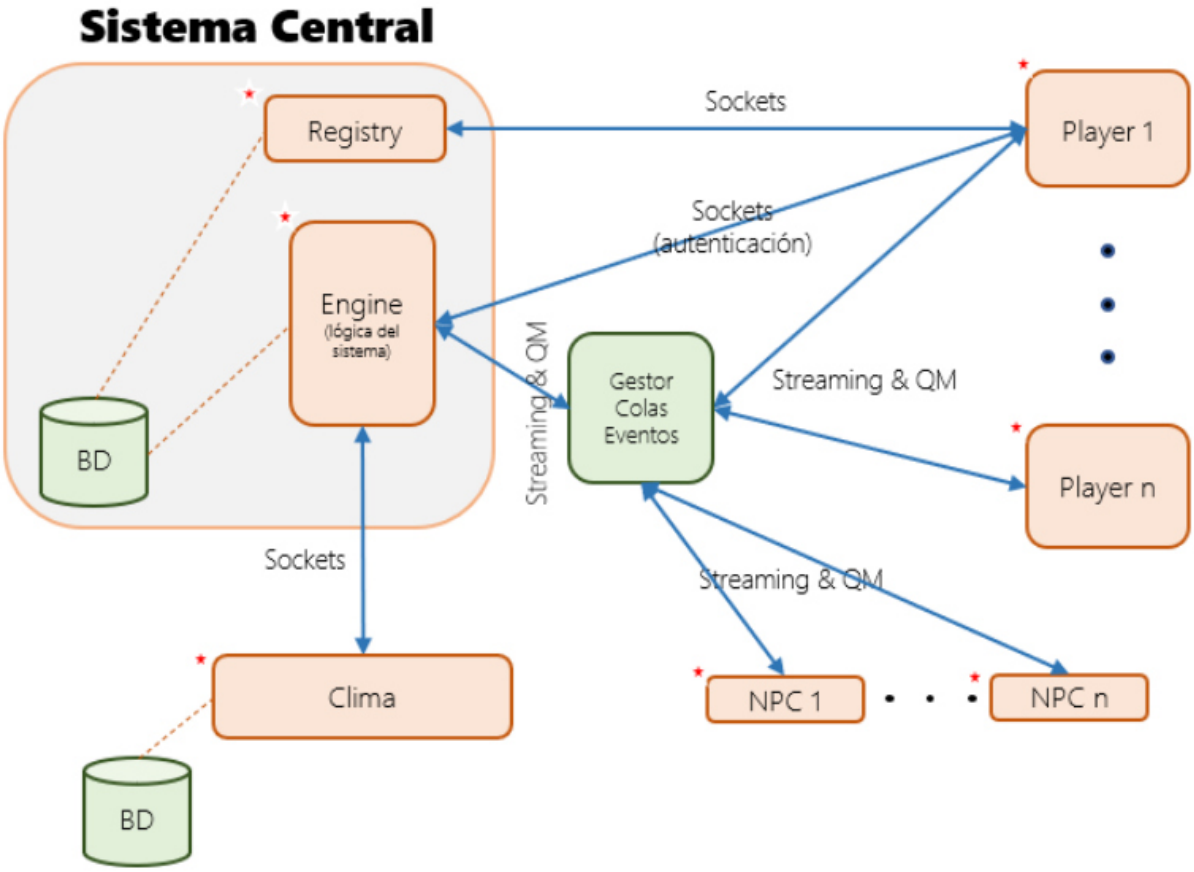
Sistemas distribuidos

Práctica no guiada: Sockets, Streaming de eventos, Colas y Modularidad



Autores:

Vadym Formanyuk: X5561410X

Alicja Wiktoria Hyska: Y3739837M

Índice

Resumen

En el siguiente documento explicaremos a detalle la estructura y las partes funcionales del proyecto, cuyo objetivo es aplicar el uso de la tecnología de comunicación aplicable a los sistemas distribuidos.

El proyecto consta de varias partes que incluyen el uso de bases de datos, tecnología de comunicación sockets, streaming y gestor de colas, aparte de la propia lógica del sistema y del juego en cuestión. A continuación, se explicarán todas las partes del sistema distribuido según su orden y lógica.

El lenguaje de programación utilizado para la implementación de la práctica es Python.

1. Sistema Central

El sistema central consta de las siguientes partes:

- Registry: el registro de los jugadores en el núcleo.
- Engine: la lógica funcional del juego.
- Database: la base de datos del sistema que deberá contener el mapa y la lista de los jugadores y sus datos.

1.1. AA_Registry

La función principal de Registry es el registro de los jugadores. En este módulo se utiliza la implementación de los sockets, importando la librería socket de python, para permitir el envío de la información que requiere. Aquí podemos ver como se ha iniciado el servidor (socket):

```
def iniciar_ServidorRegistry():

    direccion = (ip, puerto_escucha)
    server.bind(direccion)
    server.listen()
    print(f"[AA_Registry] Servidor a la escucha en {direccion}")

    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handle_player, args=(conn, addr))
        thread.start()
```

Tanto en este módulo, como en los demás, los datos de entrada se reciben a partir de un archivo json. En este caso, recibirá un puerto de escucha. Para su función, también necesitará datos como ip del servidor, el usuario, la contraseña y la base de datos en la que cargar el jugador.

Veremos que en los siguientes módulos se llamará al Registry con una cadena específica para que realice una operación que guardará en la base de datos, que puede ser crear un jugador o modificar sus datos:

- Insertar jugador.

```
op == "crear_perfil":
 if conexion_db.insertar_Jugador(datos["alias"],datos["password"],datos["nivel"],datos["ef"],datos["ec"],datos["posicion"]):
    conn.send(f"Jugador creado.".encode('utf-8'))
 else:
    conn.send(f"ERROR: Jugador no creado.".encode('utf-8'))
```

- Modificar jugador.

```
op == "editar_perfil":
 if conexion_db.modJugador(datos["alias"],datos["password"],datos["nivel"],datos["ef"],datos["ec"],datos["posicion"]):
    conn.send(f"Jugador modificado.".encode('utf-8'))
 else:
    conn.send(f"ERROR: Jugador no modificado.".encode('utf-8'))
```

Las funciones insertar_Jugador() y modJugador están implementadas en la clase db.py, de la que hablaremos más adelante. En resumen, le pasamos los parámetros que se deben guardar del jugador:

- Alias.
- Contraseña.
- Nivel.
- Efecto ante el frío.
- Efecto ante el calor.

Y la clase se encarga de pasar la información a la base de datos.

1.2. AA_Engine

La clase Engine se ocupa de la lógica fundamental del juego. Implementa la comunicación por sockets para enviar información a otros procesos y hacer el uso del mismo, de la misma manera que Registry. Del archivo de argumentos json obtiene el ip, el usuario, la contraseña y root de la base de datos para funcionar con la misma. De información imprescindible para Engine recibe puerto de escucha y el máximo número de jugadores que pueden entrar en una partida.

1.2.1. Base de Datos

Se conecta con la base de datos de la misma forma que vimos en Registry, a continuación comprueba que los datos del jugador que entra existen y en el caso afirmativo lo añade a la partida. Da mensaje de error siempre que algún dato no corresponga a los que hay en la base de datos:

```
conexion_db = db()

if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):

    if globals.PARTIDA_DISPONIBLE:
        if len(conexion_db.logeo(datos['alias'],datos['password'])) > 0: #se ha encontrado una cuenta con esos datos
            conn.send(f"Datos de login correctos.".encode('utf-8'))
            JUGADORES_PARTIDA.append(datos['alias']) #añado al jugador a la cola de espera
            print(f"[AA_Engine SERVER] El jugador '{datos['alias']}' se ha unido a la partida.")
        else:
            conn.send(f"ERROR: Datos de login incorrectos o el jugador con ese alias no existe.".encode('utf-8'))
    else:
        print(f"PARTIDA_DISPONIBLE:{globals.PARTIDA_DISPONIBLE}")
        conn.send(f"ERROR: No hay ninguna partida disponible.".encode('utf-8'))

    ##CIERRO CONEXIÓN
    conexion_db.closeCommunication()
```

1.2.2. Comunicación por sockets

Una parte importante que construye la lógica del juego es la conexión mediante sockets con el servidor de clima. En este caso, el Engine sería el cliente que necesita la información del servidor de clima, necesitará 4 ciudades (una por cada cuadrante del mapa), por lo que recorre un bucle dentro del cual recibe una ciudad y la va añadiendo a una lista. Una vez obtenidas las ciudades, se desconecta del servidor de clima y cierra la conexión:

```python
def connect_To_AA_Weather(): #Obtengo ciudad random con cada conexión
    ciudades = []
    try:
        ip_weather = args_json['AA_Weather'][0]['ip']
        puerto_a_conectar = args_json['AA_Weather'][0]['puerto_escucha']
        direccion = (ip_weather, puerto_a_conectar)

        client.connect(direccion)
        #print (f"Establecida conexión a AA_Weather")

        # Obtengo 4 ciudades
        for i in range(4):
            send_To_AA_Weather("Dame ciudad")
            server_response = client.recv(2048).decode('utf-8')
            #print("Recibo del Servidor: ", server_response)
            ciudades.append(server_response)

        ## MANDO MENSAJE PARA DESCONECTARME
        send_To_AA_Weather("Desconectar")
        #print("AA_Engine desconectado de AA_Weather")

        client.close()
    except Exception as error:
        if "Errno 111" in str(error):
            print("ERROR: El servidor de AA_Weather no está funcionando.")
        else:
            print(error)

    return ciudades
```

1.2.3. Implementación de Kafka

Otra parte importante del Engine es el gestor de colas implementado mediante la tecnología Apache Kafka, que se explicará más adelante con más detalle. En resumen, se trata de un sistema productor-consumidor, en el cual el productor está constantemente enviando mensajes al consumidor, el cual recibe y lee los mensajes. En este caso, cuando empieza la partida los modulos Jugador y Engine se comunican bajo un mismo topico y se van filtrando mensajes en función de su nombre:

```python
producer = KafkaProducer(bootstrap_servers=[server_kafka],
                         value_serializer=lambda x:
                         dumps(x).encode('utf-8'))

consumer = KafkaConsumer(
    topico_partida,
    bootstrap_servers=[server_kafka],
    auto_offset_reset='latest',
    enable_auto_commit=True,
    value_deserializer=lambda x: loads(x.decode('utf-8')))

#Envio a los players el mapa inicial de la partida
producer.send(topico_partida, f"inicio@{JUGADORES_PARTIDA_ALIAS}")
##enviar a cada jugador el alias del mapa suyo
producer.send(topico_partida, f"mapa@{mapa.getArray()}")
```

Aquí ponemos ver como es la definición de un productor y un consumidor, y como ya hemos mencionado, filtrará los mensajes según el nombre que corresponda:

```
for message in consumer:
```

- Movimiento:

```
if "movimiento@" in message.value:
```

- Npc:

```
if "npc@" in message.value:
```

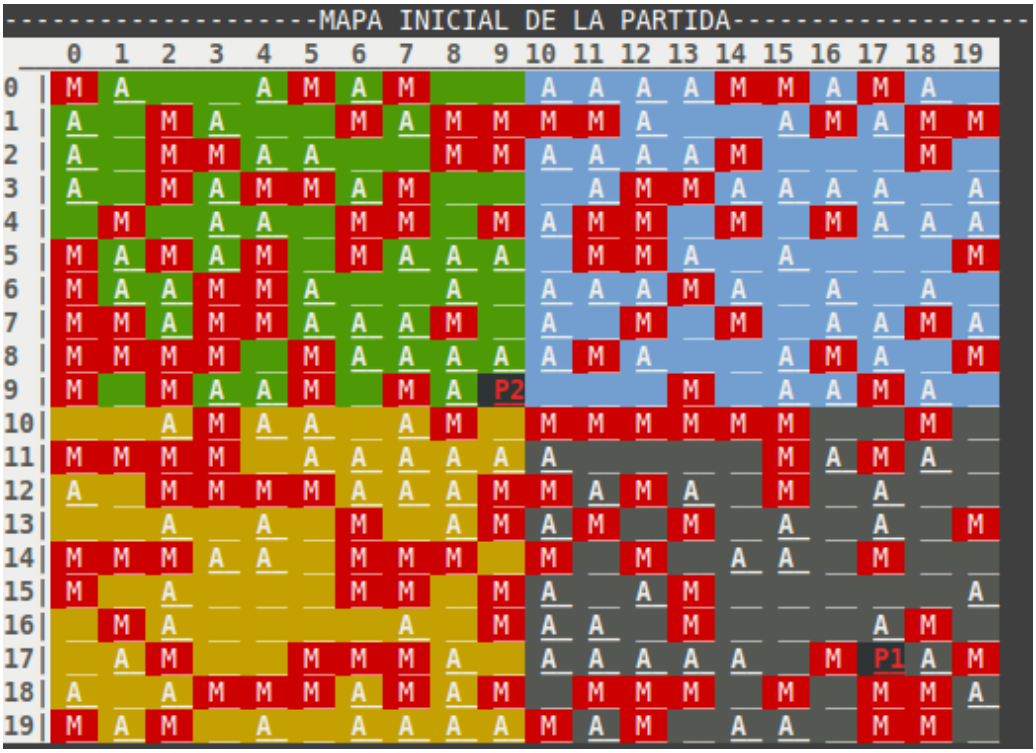
- Desconectar:

```
if "desconectar@" in message.value:
```

El filtrado de mensajes por su nombre se encuentra en la función principal de la clase Engine: *"comenzarPartida()"*, en la qeu se programa toda la logica del mismo.

1.2.4. Lógica del juego

Una vez entendido el filtrado de mensajes, entraremos con más detalle en la opción de movimiento. Cabe mencionar que esta fucnión hace referencia a un objeto imprescindible para el juego: el Mapa. Para ello, hemos creado una clase aparte que lleva el diseño del mapa sobre el cual se actúa la partida. La clase '*Mapa.py*' contiene el objeto Mapa, al que convocaremos creando el objeto donde queramos utilizarlo. Por ejemplo: mapa = Mapa(). El diseño del mismo es el siguiente:



Un mapa puede contener:

- Minas (M): que matan al jugador al pisarlas.

```
if celda == "M": # MINA
    if ciudad_actual != ciudad_movimiento: setLevel_Clima_Movimiento(alias,ciudad_movimiento,JUGADORES_PARTIDA_ALIAS)

    mapa.setCelda(pos_nueva_x,pos_nueva_y, "0") #pongo la mina a 0
    mapa.setCelda(int(pos_actual[0]),int(pos_actual[1]), "0") #pongo mi pos actual a 0
    producer.send(topico_partida, f"mapa@{mapa.getArray()}")
    producer.send(topico_partida, f"info@¡'{alias}'|'{alias_mapa}' ha muerto a causa de una mina!")
    print(f"El jugador '{alias}'|'{alias_mapa}' pisó una MINA. Ha sido eliminado.")
    JUGADORES_PARTIDA.remove(alias)
    JUGADORES_PARTIDA_ALIAS = [tup for tup in JUGADORES_PARTIDA_ALIAS if tup[0] != alias] #todas menos los que tienen ese alias
    #actualizo base de datos con su nueva posición
    jugador_setDBPosNueva(alias,pos_nueva_x,pos_nueva_y)

    #SOLO QUEDA 1 JUGADOR DESPUÉS DE QUE ESTE EXPLOTE. AL QUE QUEDE LO DECLARO GANADOR
    if len(JUGADORES_PARTIDA_ALIAS) == 1:
        alias_ganador = JUGADORES_PARTIDA_ALIAS[0][0]
        aiias_mapa_ganador = JUGADORES_PARTIDA_ALIAS[0][1]
        print(f"Solo queda 1 jugador en el mapa. Ha salido vencedor: {alias_ganador}")
        producer.send(topico_partida, f"info@¡'{alias_ganador}'|'{aiias_mapa_ganador}' ha ganado la partida!")
```

- Alimentos (A): que suben el nivel del jugador.

```
if celda == "A":# ALIMENTO
    if ciudad_actual != ciudad_movimiento: setLevel_Clima_Movimiento(alias,ciudad_movimiento,JUGADORES_PARTIDA_ALIAS)

    mapa.setCelda(pos_nueva_x,pos_nueva_y, alias_mapa) #me pongo en la nueva pos
    mapa.setCelda(int(pos_actual[0]),int(pos_actual[1]), "0") #pongo mi pos anterior a 0
    nivel, nivel_nuevo = jugador_addLevel(alias,JUGADORES_PARTIDA_ALIAS ) # le subo un nivel
    producer.send(topico_partida, f"info@¡'{alias}'|'{alias_mapa}' gana un nivel ({nivel}->{nivel_nuevo})!")
    print(f"nivel:{nivel}, nivel_nuevo:{nivel_nuevo} ")

    #actualizo base de datos con su nueva posición
    jugador_setDBPosNueva(alias,pos_nueva_x,pos_nueva_y)
```

- Otros jugadores (P): con los que tiene que luchar al cruzarse.

```
if "P" in celda:#se encuentra un jugador, luchan
    #Obtengo el alias del jugador en base a su alias del mapa (P1,P2,etc)

    for jugador in JUGADORES_PARTIDA_ALIAS: #[('testplayer', 'P1', 'nivel:37')]
            if jugador[1] == celda: #('testplayer', 'P1', 'nivel:37')
                enemigo_alias = jugador[0]
                enemigo_nivel = jugador_getNivel(enemigo_alias)
                if mi_nivel == enemigo_nivel: #EMPATE
                    producer.send(topico_partida, f"info@¡'{alias_mapa}' y '{celda}' han luchado. EMPATE!")
                else:
                    if mi_nivel > enemigo_nivel: #GANO YO
                        if ciudad_actual != ciudad_movimiento: setLevel_Clima_Movimiento(alias,ciudad_movimiento,JUGADORES_PARTIDA_ALIAS)

                        mapa.setCelda(pos_nueva_x,pos_nueva_y, alias_mapa) #voy a la casilla del enemigo derrotado
                        mapa.setCelda(int(pos_actual[0]),int(pos_actual[1]), "0") #pongo mi pos anterior a 0
                        producer.send(topico_partida, f"info@¡'{alias_mapa}' y {celda} han luchado. GANA '{alias_mapa}'!")

                        #elimino al jugadro contrario del las listas
                        JUGADORES_PARTIDA.remove(enemigo_alias)
                        JUGADORES_PARTIDA_ALIAS = [tup for tup in JUGADORES_PARTIDA_ALIAS if tup[0] != enemigo_alias] #nuevas lista = todos
                        #actualizo base de datos con su nueva posición
                        jugador_setDBPosNueva(alias,pos_nueva_x,pos_nueva_y)

                        #SI SOLO QUEDA 1 JUGADOR DESPUÉS DE QUE ESTE SALGA. AL QUE QUEDE LO DECLARO GANADOR
                        if len(JUGADORES_PARTIDA_ALIAS) == 1:

                            alias_ganador = JUGADORES_PARTIDA_ALIAS[0][0]
                            aiias_mapa_ganador = JUGADORES_PARTIDA_ALIAS[0][1]
                            print(f"Solo queda 1 jugador en el mapa. Ha salido vencedor: {alias_ganador}")
                            producer.send(topico_partida, f"info@¡'{alias_ganador}'|'{aiias_mapa_ganador}' ha ganado la partida!")

                    else: #mi_nivel < enemigo_nivel GANA ENEMIGO
                        mapa.setCelda(int(pos_actual[0]),int(pos_actual[1]), "0") #pongo mi pos actual a 0
                        producer.send(topico_partida, f"info@¡'{alias_mapa}' y {celda} han luchado. GANA '{enemigo_alias}'!")
```

- Los npc (numero): en el caso de que un player se cruze con un npc, también tendrá que luchar, sólo ganaría si tiene un nivel superior al de npc.

```
try:
    print("NPC hace un movimiento.")
    #producer.send(topico_partida, f"NPC@{posActual[0]},{posActual[1]}@{POS_X},{POS_Y}@{nivel}")
    msg = message.value.split("@")
    posActual = [msg[1].split(",")[0] , msg[1].split(",")[1]]
    posMovimiento = [msg[2].split(",")[0] , msg[2].split(",")[1]]
    mi_nivel = msg[3]

    celda = mapa.getCelda(posMovimiento[0], posMovimiento[1])
    if "P" in celda: #he encontrado un player en mi prox movimiento
        #Busco el nivel del player
        for player in JUGADORES_PARTIDA_ALIAS:
            print(f"entro en elfor {player}")
            if player[1] == celda: #he encontrado al player
                print("entro en jugador encontrado")
                nivel_player = player[2].split(":")[1]

                if int(mi_nivel) > int(nivel_player): #elimino al player

                    alias_player = player[0]

                    mapa.setCelda(posActual[0], posActual[1], "0") #pongo un 0 en la casilla del npc actual
                    mapa.setCelda(posMovimiento[0], posMovimiento[1], mi_nivel) #pongo al npc en la nueva pos
                    producer.send(topico_partida, f"info@¡'{alias_player}' y un NPC han luchado. GANA EL NPC") #informo d

                    #elimino al jugadro de las listas
                    JUGADORES_PARTIDA.remove(alias_player)
                    JUGADORES_PARTIDA_ALIAS = [tup for tup in JUGADORES_PARTIDA_ALIAS if tup[0] != alias_player] #nuevas

                    if len(JUGADORES_PARTIDA) == 1:
                        producer.send(topico_partida, f"PARTIDA TERMINADA")

                if int(mi_nivel) < int(nivel_player): #me eliminan
                    mapa.setCelda(posActual[0], posActual[1], "0") #pongo un 0 en la casilla del npc actual
                    producer.send(topico_partida, f"NPC_eliminado@{posMovimiento[0]},{posMovimiento[1]}")
```

- Casillas en blanco: por las que se puede mover el jugador.

```
if celda == "0":
    if ciudad_actual != ciudad_movimiento:
        setLevel_Clima_Movimiento(alias,ciudad_movimiento,JUGADORES_PARTIDA_ALIAS)

    mapa.setCelda(pos_nueva_x,pos_nueva_y, alias_mapa) #voy a la nueva casilla
    mapa.setCelda(int(pos_actual[0]),int(pos_actual[1]), "0") #pongo mi pos anterior a 0

    #actualizo base de datos con su nueva posición
    jugador_setDBPosNueva(alias,pos_nueva_x,pos_nueva_y)
```

Volviendo a la lógica del juego, para que un jugador pueda realizar el movimiento, le enviará un mensaje a Engine que contendrá la cadena 'movimiento@' junto con el movimiento a realizar. Los movimientos se realizan de la siguiente manera: "WASD" para moverse a una dirección de frente y "QZEX" para continuar en diagonal. El filtrado de los posibles movimientos y su implementación viene programada en la función *"GetPosNueva()"* que es llamada al principio de la opción del mensaje de movimiento. Otras funciones necesarias para la implementación de la lógica del juego:

- *jugador_getNivel(alias)*: devuelve el nivel del jugador, el cual obtiene de la base de datos.
- *jugador_addLevel(alias,JUGADORES_PARTIDA_ALIAS)*: suma una unidad al nivel del jugador.
- *jugador_setDBPosNueva(alias,pos_nueva_x,pos_nueva_y)*: carga la posición nueva (después de que el jugador haya realizado algún movimiento) en la base de datos.
- *jugador_getPosActual(alias)*: devuelve la posición actual del jugador, que obtiene de la base de datos.
- *jugadoresFromDB()*: carga a los jugadores desde la base de datos en una lista de arrays.
- *set_Pos_Random_Jugadores()*: asigna una posición aleatoria al jugador.
- *set_EF_EC_Random_Jugadores()* asigna el efecto frío y el efecto calor aleatorio al jugador.
- *setLevel_Clima_Movimiento(alias,ciudad_movimiento,JUGADORES_PARTIDA_ALIAS)*: asigan un nuevo nivel al jugador si éste cambia de ciudad (se mueve del cuadrante en el mapa).

- *setLevel_Clima_JugadoresPos(ciudades)*: calcula el nivel de los jugadores dependiendo del clima de área en el que se encuentren.
- *aux_getCiudad(ciudades,pos_x,pos_y)*: devuelve la ciudad en la que se encuentra el jugador.
- *menu()*: menú de Engine de entrada al juego.

1.3. Database

La base de datos es un elemento muy importante en la estructura del proyecto. Hemos hablado como se conecta todo el sistema central con la propia base de datos, pero ahora entraremos en más detalle.

1.3.1. Clase *db.py*

Ya hemos mencionado la clase que creamos para los datos *"db.py"*, es la clase encargada de abrir la comunicación con la base de datos y crear todas las tablas que ésta contiene con la información que se desea almacerar. Para la conexión con la base de datos:

```
def openCommunication(self,ip,user,passwd,db):
    try:
        self.conn = psycopg2.connect(
            database=db,
            user=user,
            password=passwd,
            host=ip
        )
        #Open cursor to perform database operations
        self.cur = self.conn.cursor()

        #print("Conexión con la base de datos establecida. \n")

        return True
    except Exception as error:
        print("No es posible conectarse a la base de datos. ¿Está activa? \n" )
        return False #No hay comunicación con la base de datos

#Ceerar comunicación con la base de datos
def closeCommunication(self):
    self.conn.commit()
    self.cur.close()
    self.conn.close()
```

Y opera dentro de ella de forma que crea las tablas para almacenar la información sobre jugadores y ciudades. Dentro insertaría todos los elementos que contiene cada parte con su función correspondiente:

- Insertar Jugadores: es la parte con la que se conectaría Registry para almacenar al jugador en la base de datos.

```
def insertar_Jugador(self, alias,password,nivel,ef,ec,posicion):
    try:
        self.cur.execute(f"""insert into jugadores ("alias","password","nivel","ef","ec","posicion") values ('{alias}','{password}',{nivel},{ef},{ec},'{posicion}
        self.conn.commit() #guardo la info
        print(f"Jugador {alias} insertado")
        return True
    except Exception as error:

        self.conn.rollback() #para que no dé error current transacion is aborted si intento insertar múltiples y el primero tira excepción
        if "unique constraint" in str(error):
            print(f"ERROR: No es posible insertar jugador. El jugador con el alias '{alias}' ya existe")
        else:
            print("#######[Inicio excepción]#######")
            print(error)
            print("#######[Fin excepción]#######")
        return False
```

- Insertar Ciudades: es la parte con la que se conectaría el servidor de clima para almacenar una ciudad en la base de datos.

```
def insertar_Ciudad(self, ciudad,temperatura):
    try:
        self.cur.execute(f"""insert into ciudades ("ciudad","temperatura") values ('{ciudad}','{temperatura}')""")
        self.conn.commit() #guardo la info
        print(f"Ciudad '{ciudad}' insertada.")
        return True
    except Exception as error:

        self.conn.rollback() #para que no dé error current transacion is aborted si intento insertar múltiples y el
        if "unique constraint" in str(error):
            print(f"ERROR: No es posible insertar ciudad. La ciudad con el nombre '{ciudad}' ya existe.")
        else:
            print("#######[Inicio excepción]#######")
            print(error)
            print("#######[Fin excepción]#######")
        return False
```

Dicha clase también contiene todas las funciones que operan sobre las tablas, como por ejemplo, las modificaciones de los elementos que contiene el jugador y todos los getters para obtener dicha información, o las inserciones de los jugadores y de las ciudades. En su función principal, debe abrir la conexión con la base de datos para posbilizar su despliegue:

```
iniciarDB():

#CREO OBJETO PARA LA CONEXIÓN CON LA BASE DE DATOS
conexion_db = db()

##ABRO CONEXION
if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):

    #OPERACIONES
    crearTablas(conexion_db)
    insertarCiudades(conexion_db)
    conexion_db.insertar_Jugador("vadim","pass",1,2,2,"2,2")
    conexion_db.insertar_Jugador("alicia","pass",1,2,2,"5,2")

    #CIERRO CONEXIÓN
    conexion_db.closeCommunication()
```

1.3.2. Despliegue de la base de datos

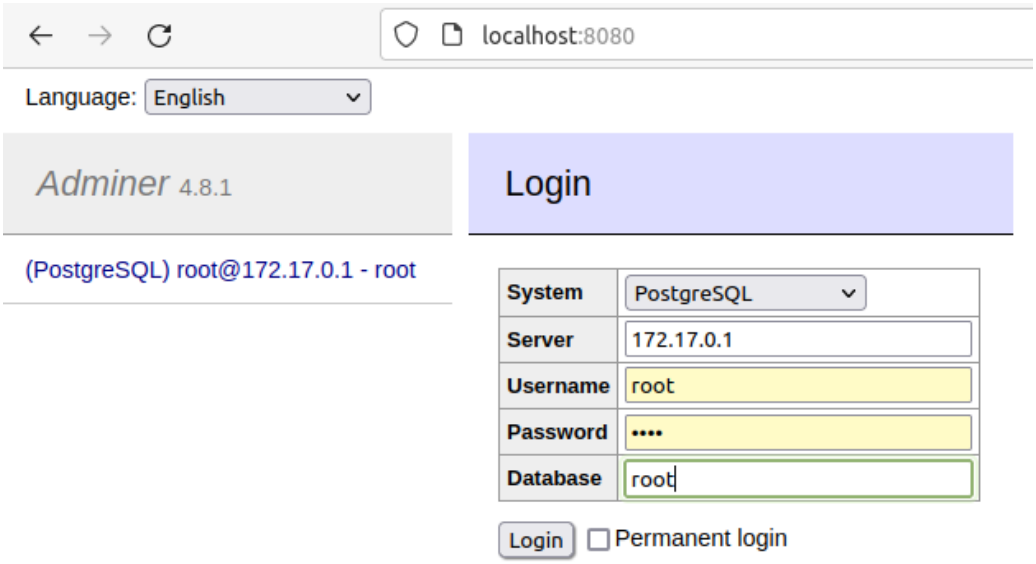
Antes de desplegar la base de datos, cabe mencionar que toda funcionamiento del programa se basa en los contenedores de Docker, nuestro bloque de la construcción principal de la aplicación. para ello, tenemos un elemento importante: *"docker-compose.yml"*, que se encarga de levantar dichos contenedores, tanto para la base de datos, como apache kafka y zookeeper.Por lo que lo primero que hay que hacer para desplegar el programa es levantar dichos contenedores con el siguiente comando:

```
sudo docker-compose up
```

El elemento que contienen dentro de docker-compose para la base de datos es:

```
postgres:
  image: postgres
  container_name: 'postgres'
  restart: always
  environment:
    POSTGRES_PASSWORD: root
    POSTGRES_USER: root
    POSTGRES_DB: root #nombre de la db que se creará al iniciar adminer por primera vez
```

A continuación, podemos ejecutar nuestra clase encargada de la base de datos. La base de datos utilizada por nosotros es PostgreSQL, y para acceder a ella, una ver que estén levantados los contenedores, tenemos que acceder al localhost con el puerto que indica docker-compose y loguearse con su IP que también deberá especializar:



Los demás elementos de entrada vienen en el archivo de argumentos, así se podrá acceder a la base de datos:

```
"db":[
  {
    "SERVIDOR_IP" : "172.17.0.1",
    "USUARIO" : "root",
    "CONTRASENYA" : "root",
    "BASE_DE_DATOS" : "root"
  }
],
```

Allí podremos visualizar todos los elementos que han sido cargados en la base de datos, como la lista con los jugadores y con las ciudades.

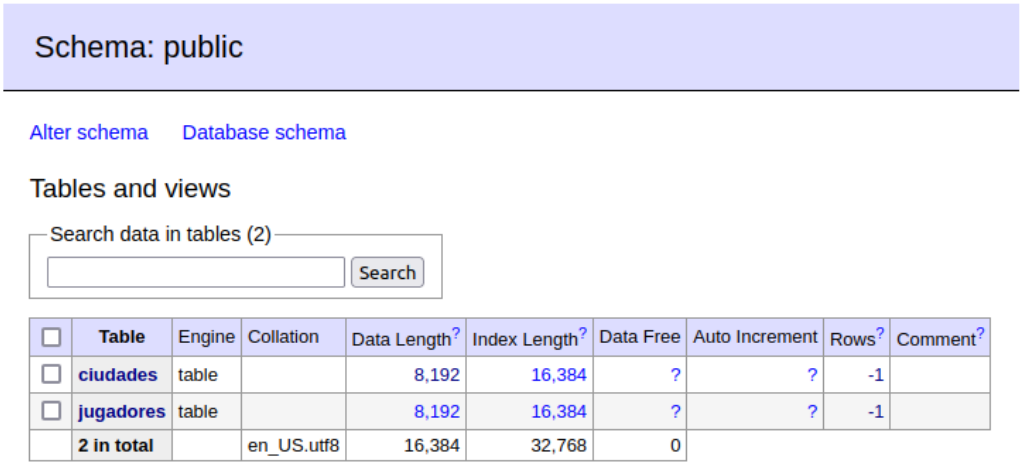


	Table	Engine	Collation	Data Length	Index Length	Data Free	Auto Increment	Rows	Comment
☐	ciudades	table		8,192	16,384	?	?	-1	
☐	jugadores	table		8,192	16,384	?	?	-1	
	2 in total		en_US.utf8	16,384	32,768	0			

2. Servidor de clima

Ya hemos hablado anteriormente del servidor de clima, es un elemento importante para el sistema central Engine. Su estructura interna está programada en la clase *"AA_Weather.py"*. Para conectar con Engine, aplica el uso de la tecnología de comunicación sockets, de la misma manera que en los ejemplos anteriores.

```python
def iniciar_ServidorWeather():

    direccion = (ip, puerto_escucha)
    server.bind(direccion)
    server.listen()
    print(f"[AA_Weather] Servidor a la escucha en {direccion}")

    while True:
        conn, addr = server.accept()
        thread = threading.Thread(target=handle_AA_Engine, args=(conn, addr))
        thread.start()
```

Del archivo de argumentos, necesitaría la información necesaria para conectarse con la base de datos, y un puerto de escucha, ya que una vez empezado el juego, el servidor queda a la espera de una 'petición' por parte del Engine.

```python
while True:

    msg_length = conn.recv(64).decode('utf-8')

    if msg_length:

        msg_length = int(msg_length)
        msg = conn.recv(msg_length).decode('utf-8')
        print(f" He recibido de AA_Engine [{addr}] el mensaje: {msg}")

        if msg == "Dame ciudad":

            ciudades = []

            #ME CONECTO A LA BASE DE DATOS
            conexion_db = db()
            if(conexion_db.openCommunication(SERVIDOR_IP , USUARIO , CONTRASENYA, BASE_DE_DATOS) == True):

                #OPERACIONES
                ciudades = conexion_db.getTabla("ciudades")
                ##CIERRO CONEXIÓN
                conexion_db.closeCommunication()

            if not len(ciudades):
                print("ERROR: No hay ciudades en la base de datos")
            else:
                ciudad_random = random.choice(ciudades)
                print("Ciudad elegida de forma random: ", ciudad_random)
                conn.send(f"{ciudad_random}".encode('utf-8')) #envio a AA_Engine la ciudad elegido de forma random
        else:
            if msg == "Desconectar":
                print("--------- AA_Engine se ha desconectado.----------")
                break
            else:
                conn.send(f"ERROR: No me has pedido una ciudad en el mensaje.".encode('utf-8'))

conn.close()
```

Cuando recibe el mensaje adecuado de Engine, crea una lista de arrays que ocupa con las ciudades que hay en la base de datos (apartado Database). Más bien, se carga la tabla entera y devuelve una ciudad aleatoria disponible de dentro de la tabla. Si el mensaje contiene la cadena desconectar, significa que Engine se ha desconectado, por lo que cierra la conexión.

3. Gestor de colas y streaming de eventos

Ya hemos explicado en el caso del engine cómo y para qué se implementa la tecnología de Apache Kafka para gestor de colas y streaming de eventos. Consiste en el productor envía constantemente los mensajes al consumidor, que a su vez recibe los mensajes. En esta práctica, los módulos de implementación de NPC's, de Players y Engine se comunican bajo el mismo

tópico en el momento en que emoieza la partida (se encuentra en el archivo de argumentos json):

```
],
"topicos":[
  {
  "topico_partida":"partida"
  }
]
```

Y así se van filtrando los mensajes según su nombre (explicado en el apartado *1.2.3 Implementación de Kafka*). La función del productor sería enviar a los jugadores al mapa inicial de la partida, mientras que el consumidor leería el mensaje que se le pasa al Engine y funcionaría en base a la imformación obtenida.

- Argumentos Kafka:.

```
topico_partida = args_json['topicos'][0]['topico_partida']
server_kafka = args_json['otros'][0]['ip_kafka'] + ":"+ args_json['otros'][0]['puerto_kafka']
```

- Productor.

```
#Envio a los players el mapa inicial de la partida
producer.send(topico_partida, f"inicio@{JUGADORES_PARTIDA_ALIAS}")
##enviar a cada jugador el alias del mapa suyo
producer.send(topico_partida, f"mapa@{mapa.getArray()}")

for message in consumer:
```

- Consumidor.

```
consumer = KafkaConsumer(
    topico_partida,
    bootstrap_servers=[server_kafka],
    auto_offset_reset='latest',
    enable_auto_commit=True,
    value_deserializer=lambda x: loads(x.decode('utf-8')),
    consumer_timeout_ms=consumidor_timeout
)
```

4. Players & Non playable characters

El siguiente módulo presenta la clase *"AA_Player.py"*, que es la encargada de implementar la funcionalidad del jugador. En el caso del player, el usuario que jueg acon el jugador puede elegir su movimiento y la estrategia. Por el otro lado de los jugadores, creamos la clase *"AA_NPC"*, que son los jugadores que toman una posición aleatoria, no pertenecen a ningún usuario.

4.1. Jugadores

La implementación del funcionamiento de un jugador vienen en la clase Player. Más adelante, veremos como funciona el registro de los jugadores y el despliegue del juego, ahora mismo vamos a adentrarnos en las conexiones que realiza la clase.

Partimos de la primera pantalla que veremos al ejecutar el jugador: la plantilla de menu():

```python
def menu():
    option = 0
    print("¡BIENVENIDO AL JUEGO!")
    while option != 1 and option != 2 and option != 3:
        print("Seleccione una opcion:")
        print("\n")
        print("1. Crear perfil")
        print("2. Editar un perfil existente")
        print("3. Unirse a la partida")
        option = int(input())
        if option == 1:
            register() #cambiar al caso de registro
        elif option == 2:
            edit_profile() #cambiar al casi de edición
        elif option == 3:
            connnect_to_Engine()
        else:
            print("ERROR. Seleccione una opción existente.")
```

Como vemos, para cada opción realizará diferentes operaciones. En el caso de la creación del perfil, se comunicará con Registry con pase de mensajes, para que cargue al nuevo jugador en la base de datos. Hará lo mismo en el caso de modificar un perfil existente, de forma que pasará por Registry (ya vimos en la parte de Registry como filtra cada mensaje):

```python
#connect_to_Registry()
direccion = (ip_AA_Registry, puerto_escucha_AA_Registry)
client.connect(direccion)

mensaje_crear_perfil = {f"alias":alias, "password":password,"nivel":1,"ef":1,"ec":1,"posicion":1}
json_mensaje = json.dumps(mensaje_crear_perfil)

send_To_Server("crear_perfil" + "@" + json_mensaje)

server_response = client.recv(2048).decode('utf-8')
print(server_response)

send_To_Server("Desconectar")
client.close()
```

En el caso de unirse a la partida, se conectará a Engine para comenzar con la lógica del juego:

```python
# ME CONECTO POR SOCKETS A ENGINE PARA LOGEARME
direccion = (ip_AA_Engine, puerto_escucha_AA_Engine)

client.connect(direccion)

print("Introduzca su alias:")
alias = input()
print("Introduzca su password:")
password = input()
mensaje_logeo = {
    "alias":alias,
    "password":password
}
json_mensaje = json.dumps(mensaje_logeo)

send_To_Server("login" + "@" + json_mensaje)

client.settimeout(3) #para que no se quede esperando respuesta infinitamente
server_response = client.recv(2048).decode('utf-8')

## MANDO MENSAJE PARA DESCONECTARME
send_To_Server("Desconectar")
client.close()
```

Si la identificación es correcta y nada ha producido errores, se activa el productor-consumidor.

1. Productor.
   En la parte del productor, en pocas palabras, el jugador envía los movimientos a realizar (bajo el mismo topico que en engine) y engine leerá el mensaje y lo filtrará asigando la función correcpondiente.

```python
def productor(alias) -> None:
    global ELIMINADO
    global MI_ALIAS
    producer = KafkaProducer(bootstrap_servers=[server_kafka],
                    value_serializer=lambda x:
                    dumps(x).encode('utf-8'))

    op=""
    i : int = 0
    while op!= "X":

        if i > 0:
            print_comoMoverJUgador()

        op= input()
        if ELIMINADO : break #cuando nos eliminan nos salimos del menú de juego
        if op in ["w","W","s","S","a","A","d","D","q","Q","e","E","z","Z","x","X","salir","SALIR"]:

            if op == "salir" or op == "SALIR":
                producer.send(topico_partida,f"desconectar@Alias:{alias}@Player:{MI_ALIAS}") ## salir
                ELIMINADO = True
                LISTA_JUGADORES.clear()
                MI_ALIAS = ""
                break
            else: producer.send(topico_partida,f"movimiento@Alias:{alias}@Player:{MI_ALIAS}@Movimiento:{op}") ## mover
        i = i + 1

def print_comoMoverJUgador():
    print("\n@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
    print("Como mover al jugador: ")
    print("W->arriba, S->abajo, A->izquierda, D->derecha, Q->arriba-izquierda, E->arriba-derecha, Z->abajo-izquierda, X->abajo-derecha; salir-> Salir del juego")
    print("@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")
```

2. Consumidor.
   El jugador también puede tomar el rol del consumidor, de manera que recibirá los mensajes que va pasando Engine (como productor) y en función de ellos obrará de la forma correspondiente.

```python
try:
    consumer = KafkaConsumer(
        topico_partida,
        bootstrap_servers=[server_kafka],
        auto_offset_reset='latest',
        enable_auto_commit=True,
        value_deserializer=lambda x: loads(x.decode('utf-8')),
        consumer_timeout_ms=consumidor_timeout
        )

    for message in consumer:

        if "PARTIDA TERMINADA" in message.value:
            print(bg.red + "La partida fué suspendida por AA_Engine o el servidor se ha caido." + bg.rs)
            ELIMINADO = True
            break #salgo del consumidor (partida)

        if "inicio@" in message.value:

            print(bg.blue + "################################## COMIENZA LA PARTIDA ##################################" + bg.rs)
            string_players = message.value.split("@")[1] #recibo una lista de tuplas: [(alias,alias_mapa),etc]
            print(bg.black + fg.orange + f"JUGADORES PARTIDA: {string_players}" + fg.rs + bg.rs)
            LISTA_JUGADORES = eval(string_players)
            for player in LISTA_JUGADORES:
                if player[0] == alias:
                    MI_ALIAS = player[1]
                    MI_NIVEL = player[2]
                    print(bg.black + fg.orange + f"Soy el jugador '{MI_ALIAS}', Mi nivel es: {MI_NIVEL}"  + fg.rs + bg.rs)

        elif "mapa@" in message.value:

            if i == 0:
                print_comoMoverJUgador()
```

4.2. NPC

Estos tipos de jugadores consisten en que son manejadas automáticamente por el sistema. En lugar de que el usuario sea el que lo mueva, estos eligen una posición aleatoria. Los posibles movimientos para realizar están implementados en la función *"getPosNueva()"*, la cual es similar a la de engine para el movimineto del jugador.

Los NPC's toman el rol del consumidor cuando lo añadimos a la partida, ya que pasa a esperar a recibir algún mensaje en el tópico donde se produce la partida. Si el mensaje que se envía dle engine por el topico contiene NPC@, se incluirán a la partida. Cuando el mensaje contiene NPC_eliminado@, se excluirá de la partida.

```
consumer = KafkaConsumer(
    topico_partida,
    bootstrap_servers=[server_kafka],
    auto_offset_reset='latest',
    enable_auto_commit=True,
    value_deserializer=lambda x: loads(x.decode('utf-8')),
    consumer_timeout_ms=productor_timeout
    )

print("NPC creado. Esperando partida...")
#El npc se pone a escuchar si hay alguna partida empezada
for message in consumer:

    if "inicio@" in message.value or "mapa@" in message.value or "info@" in message.value or "jugadores@" in message.value:
        PARTIDA_DETECTADA = True
        print("Partida detectada.")
        break #dejo de escuchar al topico


if PARTIDA_DETECTADA:
```

Por otro lado, tomará el rol de productor cuando, una vez esté en la partida, enviará un movimiento aleatorio:

```
if PARTIDA_DETECTADA:
    print("Uniéndome a la partida...")
    #Pongo el consumidor en marcha
    Thread(target=consumidor).start()

    #creo el productor de movimientos
    producer = KafkaProducer(bootstrap_servers=[server_kafka],
                             value_serializer=lambda x:
                             dumps(x).encode('utf-8'))

    #busco un lugar de inicio random y mi nivel
    POS_X = random.randrange(0,19)
    POS_Y = random.randrange(0,19)
    nivel = random.randrange(1,9)
    #Comiento a enviar mis movimientos al engine mientras no se haya terminada la partida
    while True:
        if(PARTIDA_TERMINADA): break

        movimiento = random.choice(["W", "S", "A", "D", "Q", "E", "Z", "X"]) #selecciono un movimiento random
        posActual = [POS_X,POS_Y]
        POS_X,POS_Y = getPosNueva(posActual, movimiento) #pos_Movimiento
        #NPC @ pos_Antigua @ pos_Movimiento @ Nivel
        producer.send(topico_partida, f"NPC@{posActual[0]},{posActual[1]}@{POS_X},{POS_Y}@{nivel}")
        print(f"Movimiento:{POS_X},{POS_Y}")

        time.sleep(5) #me quedo dormido 2 sec antes de realizar otro movimiento
```

5. Despliegue del juego

Una vez conocidas todas las partes del programa en individual, vamos a desplegarlo y probar el juego para comprender del todo su funcionamiento.

5.1. Comandos

Vamos a ir por partes, ya se ha mencionado anteriormente que el programa no funcionaría sin nuestros contenedores de docker, por lo que lo primero y lo principal el levantar los contenedores:

`sudo docker-compose up`

Una vez levantados, podemos desplegar nuestra base de datos con la clase que contiene todos sus elementos (nosotros usamos python3), con ello nos crea la tabla y las ciudades:

`python3 db.py`

Lo siguiente a tener en cuenta es el servidor de clima, que permanecerá a la escucha esperando a que la aplicación Engine le pida una ciudad aleatoria y su temperatura:

`python3 AA_Weather.py`

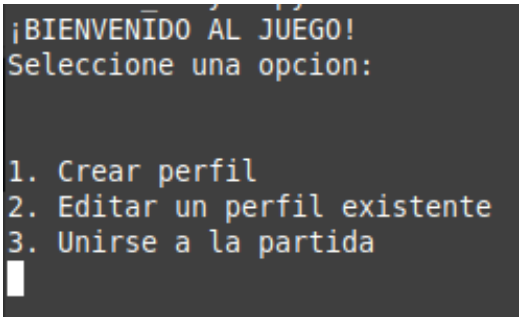
Para poder registrar a nuestro usuario (o bien modificarlo), será importante ejecutar el Registry, que al igual que el servidor de clima, se queda a la espera del player que quiera crear o modificar su perfil:

```
python3 AA_Registry.py
```

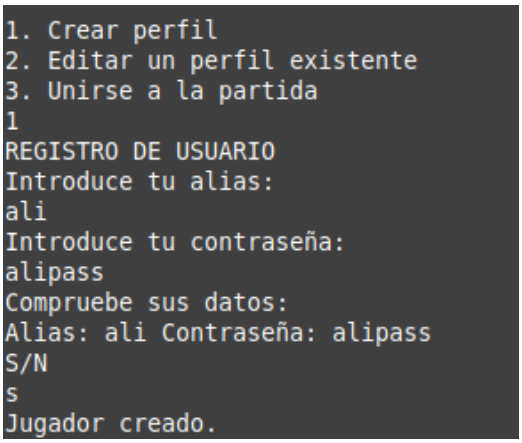
Y lo último y lo más imprescindible para empezar a jugar es ejecutar Engine y los jugadores que quieran unirse al juego. Para poder crear al jugador con nuestro alias, ejecutaremos el player de la siguiente manera:

```
python3 AA_Player.py
```

Nos debe aparecer un menú de registro, edición o la unión a la partida:



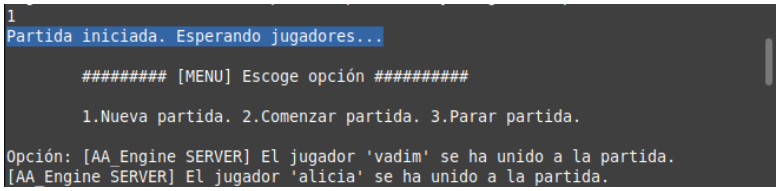
En este caso, para jugar por primera vez, vamos a crearnos un jugador, con la opción 1:



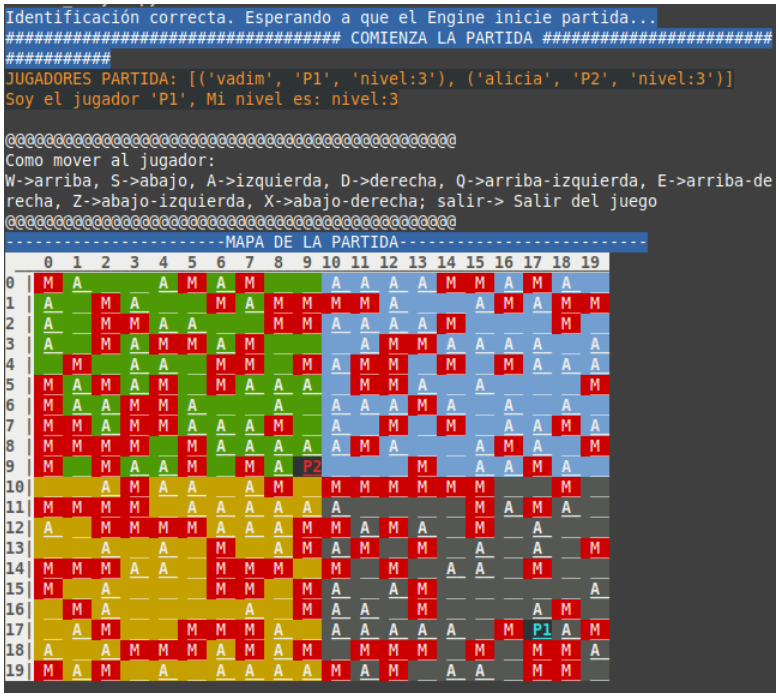
Cuando nos aparece "Jugador creado", es que el jugador se ha cargado en la base de datos sin errores. Cuando queramos comenzar la partida, elegiremos la opción 3 del menú, pero para ello tenemos que desplegar el engine, ya que es el que contiene la lógica del juego:

```
python3 AA_Engine.py
```

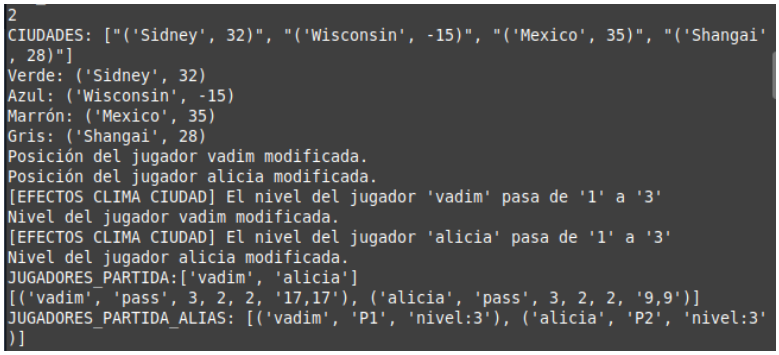
Cuando ejecutemos Engine, nos aparecerá la menú para unirse a la partita, el menú principal del juego, así que si queremos participar, escogeremos la opción 2. En este caso, como queremos iniciar una partida nueva con nuestro sjugadores creados, le daremos a la opción 1 de iniciar una partida nueva:



Una vez iniciada la partida, conectamos a los jugadores: en este punto volvemos al Player. Ya hemos mencionado que para empezar la partida el usuario debe escoger la opción 3, que llamará a la fución "connect_to_Engine()" de player.



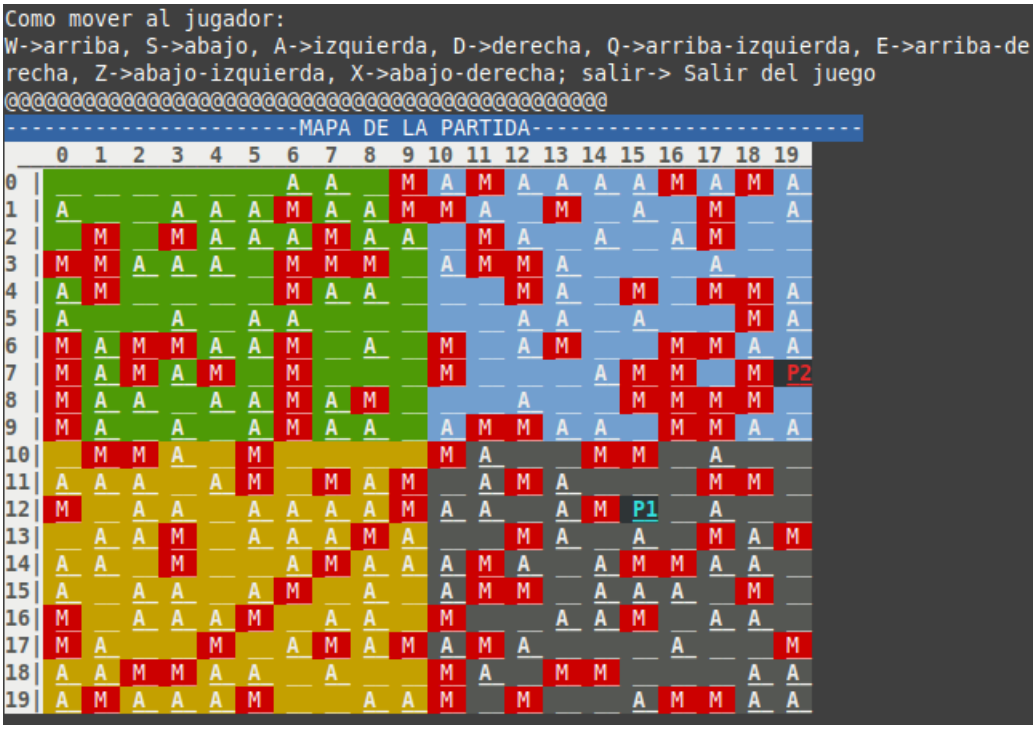
En este momento, en engine ya podemos observar como se ha conectado un jugador y el juego está listo para su prueba, por lo que comenzaremos la partida con la opción 2 del menú:



En este punto se cocentan las ciudades, el mapa es visible y la batalla puede comenzar!

5.2. ¡Que empiece el juego!

El juego funciona de forma siguiente, para ganar tienes que ser el último jugador que queda en el mapa. En la terminal de tu player, tú aparecerás de color azul, mientras que tus rivales aparecerán de color rojo, ya sean otros jugadores o los npc. Por ejemplo, en este caso somos el P1:



Durante el juego, tenemos que ir subiendo de nivel, esto se consigue:

- Comiendo alimentos (+1).
- Cambiando de ciudad (dependiendo de la ciudad).

Por lo que podremos matar a nuestro oponente sólo si entramos en la batalla con él (pasándonos a su posición) con un nivel superior a él. En caso contrario, si nuestro rival tiene más nivel, nos matará a nosotros.

Otra forma de ganar la partida es si nuestro rival se topa con una mina (entonces muere y quedamos nosotros), o se cruza con un npc con nivel superior a él (entonces morirá en la batalla con el npc). El nivel de npc no cambia, por lo que es fácil distinguirlo por su nombre.

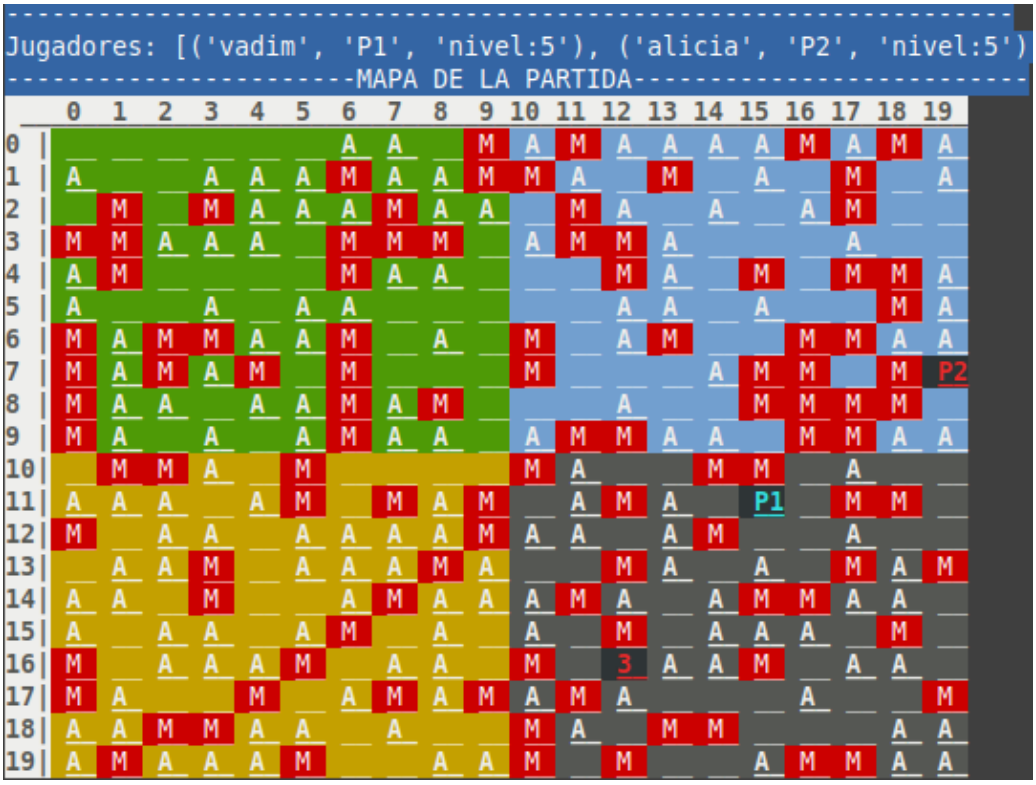
Para añadir los npc a la jugada, ejecutaremos el comando en otra terminal:

```
python3 AA_NPC.py
```

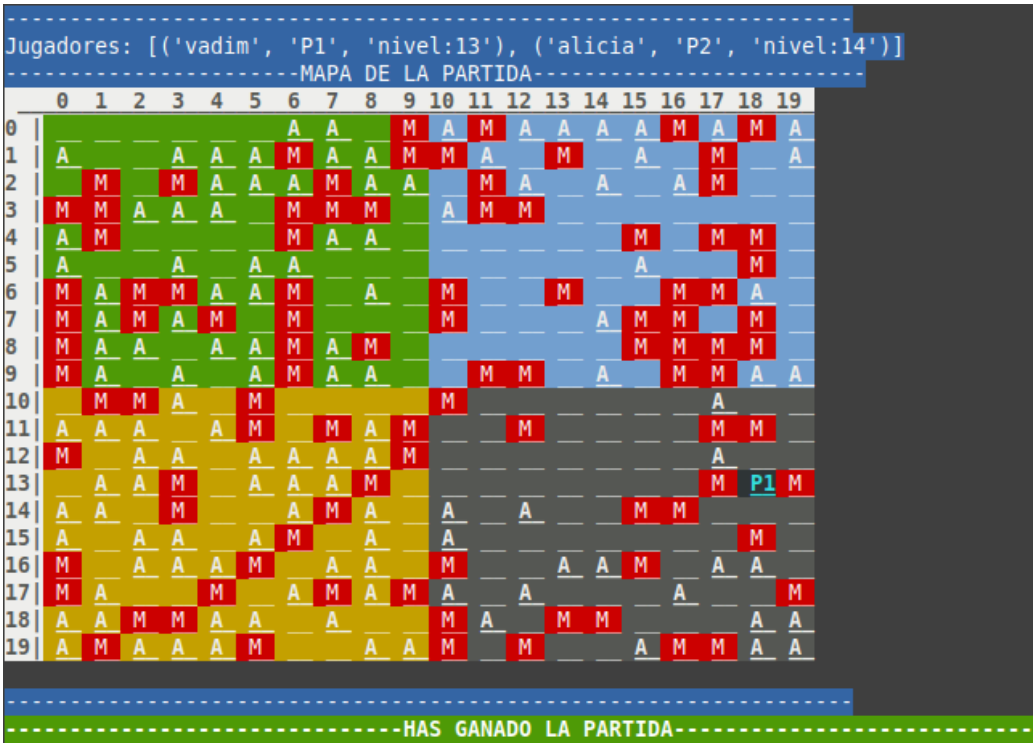
Por lo que el npc encuentra la partida y se mete en una posición aleatoria:

```
NPC creado. Esperando partida...
Partida detectada.
Uniéndome a la partida...
Movimiento:16,12
Movimiento:17,11
```

Una vez se ha metido npc, podemos observarlo en el mapa:



También podemos ver quién es cada jugador y qué nivel tiene. Una vez que un jugador gana la partida, la terminal muestra un mensaje de jugador ganador de la partida:



¡OJO! Solo tienes un minuto para realizar un movimiento, así que ¡date prisa!

Una vez conocido el funcionamiento, cada parte y cada elemento del juego, la lógica, el procedimiendo, las reglas y una guía para ganar, ¡a disfrutar!