SISTEMAS DISTRIBUIDOS

- AGAINST ALL 1 -

Javier Mellado Sánchez DNI

Álvaro Pérez Gómez 48802614H

Sumario

1. INTRODUCCIÓN

En la práctica propuesta se pretende introducir el uso de sistemas distribuidos para la realización de un juego en línea.

Para enfocarlo al círculo laboral, se pide el desarrollo multijugador que resuelva mediante el uso de mensajes, dado que es un elemento básico de la comunicación mediante los ya mencionados sistemas distribuidos.

La idea detrás de este ejercicio es la existencia de un servidor remoto que se comunique con un número indeterminado de clientes, que serán los jugadores, y permita la gestión de las peticiones que efectúa cada uno de los integrantes de la partida.

En cuanto a las reglas del juego, se dispone de un mapa cuadrado de tamaño fijo, 20 casillas de altura y anchura, donde cada casilla puede contener:

- Alimento: representado por '+', de comerlo el jugador subirá un nivel.
- Mina: dibujado con el '#', si aterrizas en una mina se acabará la partida.
- Nada: dibujado con un '·', como su nombre indica, no ocurre nada.
- Jugador o NPC: si te encuentras con uno, se disputará una batalla por la casilla.

Además, el mapa está dividido en 4 regiones cuadradas de tamaño 10, donde en función de la temperatura que tenga, los jugadores subirán o bajarán de nivel en base a la resistencia al frío y al calor que se les otorgue aleatoriamente al inicio de la partida.

La partida acabará cuando quede un único jugador en el campo, proclamándose éste el vencedor y saliendo de la partida justo después del mensaje de celebración. Otra forma de acabar sería si todos los jugadores mueren mientras existan uno o más NPCs en el terreno de juego, en cuyo caso ningún jugador humano resultaría campeón.

2. TECNOLOGÍAS EMPLEADAS

Como lenguaje para la práctica se ha decidido usar Python, pues es fácil para programar, dejando de lado problemas técnicos de lenguajes y centrándonos en el propósito de la práctica, que es la comunicación en un sistema distribuido.

2.1 BASE DE DATOS

Como base de datos se usa SQLite, que es una base de datos relacional sencilla y exportada en archivo. Su implementación en Python es simple ya que no hay que instalar ningún programa adicional.

En esta base de datos se debe guardar el mapa de la partida en todo momento, así como el alias y la contraseña de los jugadores, lo que permite tanto recuperar el estado de la partida en caso de caídas como el acceso de los usuarios al juego, ya sea para empezar una partida o para editar sus credenciales desde el menú.

Las operaciones permitidas sobre usuarios son: registrarse, dar de baja una cuenta y editar tanto tu nombre de usuario como tu contraseña, si así se quisiera.

2.2 SOCKETS

Como comunicación entre los módulos del sistema distribuido se han empleado sockets, de tipo TCP para evitar problemas de sincronización, asegurando al mismo tiempo que todos los paquetes llegan a su destino, aunque se reciban de forma más lenta, porque en este tipo de problemas la eficiencia se prioriza considerablemente frente a la velocidad.

Para facilitar su implementación se ha diseñado la clase de alto nivel *MySocket*, que facilita la creación de los sockets y por ende, la transmisión de datos.

```
class MySocket(socket.socket):
    def __init__(self, type, addr):
        super().__init__(socket.AF_INET,
            socket.SOCK_DGRAM if type == "UDP"
            else socket.SOCK_STREAM)

        if not len(addr[0]):
            self.bind(addr)
            self.listen()
            self.conn = {}
        else:
            self.connect(addr)
            self.conn = {"None":self}

    def accept(self):
        conn, client = super().accept()
        self.conn["None"] = conn
        self.conn[client] = conn
        return conn, client

    def close(self):
        self.conn, self.client = None, None
        super().close()

    def pack(data):
        packet_len = struct.pack("!I", len(data))
        return packet_len + data

    def send_msg(self, msg, client="None"):
        self.conn[client].sendall(MySocket.pack(msg.encode("utf-8")))
```

Las conexiones empiezan inicializando los sockets con *AF_INET*. Los sockets de tipo INET en Linux hacen referencia a la conexión por Internet, en donde se emplean tanto direcciones IP como puertos para establecer la comunicación entre dispositivos.

La conexión se puede producir usando dos protocolos básicos: TCP y UDP. En otras asignaturas se ha visto que TCP es más lento pero más fiable, y UDP es más rápido aunque no garantice la llegada de todos los paquetes.

De haber usado el protocolo UDP con su característico *SOCK_DGRAM*, se tendría que haber manejado manualmente la pérdida de paquetes y cómo recuperarlos, así como la llegada desordenada de los mismos. Por otra parte, los streams de TCP (*SOCK_STREAM*) permiten una conexión asíncrona sin necesidad de emplear callbacks ni protocolos de transporte a bajo nivel.

Si no se ha pasado ninguna dirección, el socket usará *bind()* para actuar de servidor y se quedará a la escucha de nuevas conexiones cliente, que se conectarán en modo TCP con su IP y un puerto al socket servidor usando el método *connect()*.

El servidor entonces acepta la petición de cada cliente y devuelve la conexión y la dirección del solicitante. Una vez aceptada, es el servidor el encargado de cerrar la conexión al cliente, ya sea por una petición del usuario o por manejo de conexiones, para lo cual se usará el método *close()*.

Con el fin de manejar la información transmitida mediante las conexiones establecidas, Python permite su almacenamiento y conversión en datos binarios. Usando *pack()* se guarda el formato en el primer parámetro (!I) y los valores almacenados en los sucesivos.

El primer parámetro es una cadena de dos caracteres, donde '!' simboliza la transmisión por conexión, por defecto a big endian, y la 'I' es que espera un entero sin signo. Este entero es la longitud de la cadena de los datos transmitidos, ya que struct sólo permite cadenas de longitud fija.

Otra operación básica de los sockets es el paso de mensajes. Usando este método el servidor puede enviar mensajes a los clientes, tales como 'Usuario creado correctamente' o el resultado de otras múltiples operaciones, sin necesidad de enviarle esa misma información a todos los clientes simultáneamente.

La característica básica de *sendall()* es que el socket remoto envía mensajes continuamente hasta que se acaba el mensaje o hasta que ocurre un error, a diferencia de *send()*, en donde la aplicación es la responsable de comprobar que se han enviado todos los datos.

```
    def send_obj(self, obj, client="None"):
        self.conn[client].sendall(MySocket.pack(pickle.dumps(obj)))

    def recv_confirmed(conn, buf_len):
        buf = b""
        while len(buf) < buf_len:
            buf += conn.recv(buf_len-len(buf))
        return buf

    def recvall(conn):
        buf_len = struct.unpack("!I", MySocket.recv_confirmed(conn,4))[0]
        return MySocket.recv_confirmed(conn, buf_len)

    def recv_msg(self, client="None"):
        return MySocket.recvall(self.conn[client]).decode("utf-8")

    def recv_obj(self, client="None"):
        return pickle.loads(MySocket.recvall(self.conn[client]))


def rundb(dbfile, query, parameters=()):
    with sqlite3.connect(dbfile) as db:
        cur = db.cursor()
        res = cur.execute(query, parameters)
        db.commit()
    return res
```

De la misma forma que con el paso de mensajes, con *send_obj()* es posible enviar un objeto entero. En esta práctica se ha empleado este método, entre otras cosas, para enviar al servidor el objeto usuario con su alias y su contraseña, con el fin de poder verificar las diversas operaciones permitidas con la base de datos que el enunciado especifica.

El mismo proceso se realiza para las operaciones con prefijo recv, que son de tipo send pero en dirección opuesta, reciben.

Por último, la base de datos efectúa las operaciones que se les pasa por parámetro. El archivo *AA_Registry* es el encargado de establecer la conexión con la base de datos. Con *MySocket* simplemente se ejecuta la operación deseada (Create, Delete, Update…) y se guardan los datos en el archivo db.db usando el método *commit()*.

2.3 GESTOR DE COLAS

Para manejar las peticiones de los movimientos de los jugadores es necesario establecer un gestor de colas que marque un orden de ejecución que impida la sobrecarga excesiva del servidor. Vamos a usar Kafka, que es el gestor pedido por el enunciado.

En nuestra arquitectura, tanto *AA_Engine* como cada uno de los jugadores son productores y consumidores al mismo tiempo: el motor consume los movimientos de los jugadores y produce el mapa resultante, y viceversa para los jugadores. En la categoría de jugadores se incluye obviamente a los NPCs, que también cuentan con conexión a Kafka.

Existen, por tanto, estos dos *topics* donde los *clusters* guardarán los datos de Kafka. Además, para que los jugadores puedan aceptar todos cada resultado del motor es necesario que todos tengan un *group_id* distinto.

Usando Docker se permite ejecutar de forma independiente la aplicación con contenedores, manteniendo la comunicación por conexiones.

Para empezar a usar Kafka se debe activar un servidor Zookeeper. Podemos asegurarnos que el servidor empieza antes que Kafka y acaba después del gestor de colas usando el archivo *docker-compose.yml* que se encuentra en la carpeta *deploy*.

En ese archivo se especifican los dos servicios empleados, siendo en nuestro caso *zookeeper* y *kafka*, con puertos 22181 y 29092, respectivamente.

En estos servicios, *zookeeper* está a la escucha en el puerto 2181, definido en el mismo contenedor, aunque está expuesto a los clientes en el mencionado 22181. Algo similar ocurre con el servicio *kafka*, que escucha en el 9092 aunque esté expuesto a las aplicaciones del host usando el 29092 en el contenedor especificado en el apartado de *KAFKA_ADVERTISED_LISTENERS* del archivo '.yml'.

Una vez configurado se pone en marcha usando el comando *sudo docker-compose up -d*:

```
apg203@LAPTOP-5QAL19FV:/mnt/c/Users/alvar/Downloads/SDestrozo-main/deploy$ sudo /etc/init.d/docker start
[sudo] password for apg203:
 * Starting Docker: docker                                                                    [ OK ]
apg203@LAPTOP-5QAL19FV:/mnt/c/Users/alvar/Downloads/SDestrozo-main/deploy$ sudo docker-compose up -d
Starting deploy_zookeeper_1 ... done
Starting deploy_kafka_1     ... done
```

Sin embargo, el gestor de Kafka no borra los datos al acabar, por lo que al iniciar la siguiente aplicación aún pueden quedar rastros de la anterior si no se eliminan. Estos restos se pueden borrar usando comandos como *sudo docker-compose down -v*.

```
apg203@LAPTOP-5QAL19FV:/mnt/c/Users/alvar/Downloads/SDestrozo-main/deploy$ sudo docker-compose down -v
Stopping deploy_zookeeper_1 ... done
Removing deploy_kafka_1     ... done
Removing deploy_zookeeper_1 ... done
Removing network deploy_default
apg203@LAPTOP-5QAL19FV:/mnt/c/Users/alvar/Downloads/SDestrozo-main/deploy$ sudo docker-compose ps
Name   Command   State   Ports
------------------------------
apg203@LAPTOP-5QAL19FV:/mnt/c/Users/alvar/Downloads/SDestrozo-main/deploy$ sudo docker-compose up -d
Creating network "deploy_default" with the default driver
Creating deploy_zookeeper_1 ... done
Creating deploy_kafka_1     ... done
apg203@LAPTOP-5QAL19FV:/mnt/c/Users/alvar/Downloads/SDestrozo-main/deploy$ sudo docker-compose ps
      Name                     Command               State                                Ports
-------------------------------------------------------------------------------------------------------------------------
deploy_kafka_1       /etc/confluent/docker/run   Up      0.0.0.0:29092->29092/tcp,:::29092->29092/tcp, 9092/tcp
deploy_zookeeper_1   /etc/confluent/docker/run   Up      0.0.0.0:22181->2181/tcp,:::22181->2181/tcp, 2888/tcp, 3888/tcp
```

3. ARQUITECTURA DE LAS APLICACIONES

Todos los servicios aportan información sobre su estado y las peticiones en forma de logs que se muestran por terminal.

3.1 AA_WEATHER

Se trata de un servicio encargado de enviar una ciudad y su temperatura asociada. Como argumento se tiene únicamente su puerto, desde el cual se queda a la escucha, esperando una solicitud proveniente de AA_Engine.

Una vez recibida la petición, enviará tantas ciudades aleatorias junto a su temperatura como el solicitante especifique, siendo en nuestro caso cuatro, que se representan en los cuadrantes 10 x 10 mencionados en la introducción. Se controla, además, que las ciudades enviadas al receptor sean distintas entre sí, aunque si hay dos o más ciudades con el mismo valor de temperatura, no es necesario cambiarla.

Como base de datos de las ciudades se ha usado un archivo CSV, con una columna que almacena las distintas ciudades escogidas y, separadas por un punto y coma, se guardan sus respectivas temperaturas.

3.2 AA_REGISTRY

Este servicio es el que se ocupa de interactuar con la base de datos principal SQLite usando una especie de API de sockets. De esta manera, es posible comunicarse con el cliente y que éste efectúe operaciones básicas sobre su perfil, como las ya mencionadas CRUD.

Como argumento se pasa su puerto por el que se quedará también a la escucha, esperando una solicitud proveniente de *AA_Engine*.

3.3 AA_ENGINE

Es el núcleo del sistema, que se encarga de crear y manejar la lógica del juego y las conexiones para que sea posible la conexión multijugador.

Usando esas conexiones, se comunica con los jugadores, de forma que les envía a todos el mapa resultante y los consecuentes mensajes, tales como ‘Fin de la partida’ o ‘Victoria’, entre otros detalles.

Su labor de comunicarse con la base de datos permite, por ejemplo, que un jugador no entre dos veces a la partida, o que se dé de baja un usuario maliciosamente sin saber su contraseña, lo que otorga cierta seguridad al sistema.

Sus argumentos son: el puerto de escucha, máximo de jugadores, IP y puerto de AA_Weather y la IP del gestor de colas (el puerto no ya que es fijo, el 29092).

Por tanto, el funcionamiento básico del motor se puede dividir en dos partes:

3.3.1 UNIÓN DE JUGADORES

En esta fase el servidor está a la escucha de nuevos jugadores que deseen unirse. Al momento de conectarse, el motor crea un hilo que permita atender al cliente.

En caso de ser un jugador humano, debe pasar el proceso de autenticación. El cliente introduce sus credenciales, lo que abrirá una conexión con la base de datos, y ésta, tras la operación, le comunica al motor si el cliente es válido o no.

Tanto si el usuario es inválido como si se ha superado el número máximo de jugadores, el motor cierra la conexión con ese cliente. Si, por el contrario, se ha podido entrar a la partida sin complicaciones, el servidor esperará a que el cliente presione listo.

Cuando en el hilo se revise que todos los jugadores estén listos, se cierra el socket servidor de *AA_Engine*, terminando con el bucle de aceptación de jugadores y por ende, empezando la partida.

3.3.2 PARTIDA

Por cada jugador se inicializa el juego y se crea un jugador en una posición aleatoria en la que no haya ni minas ni alimentos, sino un espacio en blanco. Es a través de Kafka que ese mapa inicial se puede transmitir a todos los usuarios.

Para el resto de la partida *AA_Engine* se encargará de leer los mensajes de Kafka donde se expresan los movimientos de los jugadores, respondiendo a todos con el mapa resultado del movimiento. Además, se comprueba si el resultado da como concluida la partida, finalizando el juego. De ser el caso, el motor se reiniciará para poder crear otra sala.

Para simplicidad del código se han creado tres clases dentro del motor, que permiten gestionar el estado de la partida:

La clase *Cell* controla el estado de una celda del mapa. Hay que tener en cuenta que las celdas inicialmente sólo eran una cadena con su elemento, pero de haber una confrontación entre dos jugadores y tener ambos el mismo nivel, los dos deben permanecer en la casilla, por lo que es necesario que cada celda sea multivalor, aunque en la mayor parte de los casos únicamente tengan uno.

Otra clase es *Map*, que es una lista de listas, donde cada lista pequeña tiene veinte valores para las veinte celdas que hay en una fila, mientras que la lista más grande se compone de las veinte listas, formando una matriz con posiciones fila y columna, donde en cada posición se representa una casilla monovalor o multivalor.

La última clase es *Game*, que contiene el mapa y los jugadores participantes. Al agregar un jugador, es esta clase la que le asigna una casilla inicial. Además, cada movimiento efectuado por los jugadores se asimila y opera desde el método *move()*, que permite actualizar la posición del jugador en función de la dirección que haya escogido.

Usando el método *update()*, *Game* comprueba el estado de la celda destino y le comunica al usuario lo que ha ocurrido según la lógica del juego (si se come una mina se muere, sube de nivel si come un alimento, una lucha o hasta los cambios de temperatura).

3.4 AA_PLAYER

Se limita a conectarse con *AA_Engine* para realizar diversas operaciones. Mientras no esté en una partida o esperando a jugadores, cada usuario puede crear, borrar o editar tantos perfiles como desee si inserta las credenciales correctas, conectándose a *AA_Registry*.

Al introducir los datos se envía una solicitud mediante sockets al motor con la operación que desea efectuar y se encuentra a la espera del mensaje de error o de éxito.

Una vez está a la espera de la partida, debe presionar ENTER para estar listo y, cuando empiece la partida, se recibirá un mapa por parte del motor. Llegados a este punto, el jugador introduce una tecla para moverse en una dirección y le pasa mediante hilos el resultado al motor, que como hemos visto, asimilará la coordenada y le devolverá al usuario el resultado correspondiente.

Además, el usuario se encarga de imprimir por terminal el mapa que recibe usando otro hilo desde *AA_Engine* hasta que sea eliminado o se acabe la partida.

Los argumentos que recibe son: IP y puerto de AA_Registry, IP y puerto de AA_Engine e IP del gestor de colas.

3.5 AA_NPC

El NPC es un jugador como *AA_Player*, con la distinción de que no lo controla ningún humano. El NPC contará con un nivel aleatorio al inicio de la partida, que viene dado por AA_Engine, y no se verá alterado durante el transcurso de la misma.

El NPC no puede interactuar con alimentos ni minas pero no es inmortal, ya que si un jugador entra en conflicto con él, se aplica el método *fight()* de *Game* como si de dos jugadores humanos se tratara.

Similar a un jugador, recibe un mapa y envía un movimiento cada cierto tiempo, con la salvedad de que el NPC no posee interfaz alguna, mientras que el humano sí.

Como los NPCs no se registran, su único argumento es la IP del gestor de colas.

4. ARQUITECTURA DE LAS CONEXIONES

A continuación se muestra como los clientes se deben comunicar con los módulos servidores:

4.1 CONEXIÓN CON AA_WEATHER

AA_Engine se conecta con *AA_Weather* para obtener las 4 ciudades que conforman el mapa del juego.

```
class Requests():
    @staticmethod
    def get_cities(n):
        cities = set()
        while(len(cities) < 4):
            with MySocket("TCP", ADDR__AA_WEATHER) as client:
                city = client.recv_obj()
                if city["city"] not in cities:
                    cities.add(city["city"])
                    yield city
```

Se puede apreciar que se abre un socket en el que, mediante la dirección IP y puerto contenidos en *ADDR_AA_WATHER* se establece una conexión TCP.

Al enviar la solicitud, el programa del tiempo envía un objeto *city* al motor que está formado por el nombre de la ciudad y por su temperatura, sacado directamente del archivo CSV donde está almacenado.

4.2 CONEXIÓN CON AA_REGISTRY

El jugador se conecta con el registro para operar con los datos del propio usuario, buscando un contacto entre el humano y la base de datos.

```
if confirmation == "S" or confirmation == "s":
    client = MySocket("TCP", ADDR_R)
    client.send_msg("Create")
    client.send_obj((usr,passwd))
    msg = client.recv_msg()
    printMsg(msg)
    client.close()
```

Lo primero que se hace es abrir el socket como cliente. Es posible abrirlo declarando la variable client o mediante la estructura *with MySocket(<params>) as client*, pero si se declara como en la captura hay que recordar cerrar la conexión usando *close()*.

El cliente, tras establecer la conexión, envía un mensaje con la operación que desea; crear, editar o borrar. Si se quiere crear o borrar un usuario, se envían al servidor tanto el nombre de usuario como la contraseña como un objeto, y el servidor responderá con el mensaje obtenido desde la base de datos.

```
client = MySocket("TCP", ADDR_R)
client.send_msg("Update")
client.send_obj((username,password))
client.send_obj((usr,passwd))
msg = client.recv_msg()
printMsg(msg)
client.close()
```

En cambio, si se quiere actualizar algún campo del usuario, se transmite, además del usuario y contraseña, los nuevos valores a adoptar. Recibe igualmente un mensaje indicando el éxito o el error de la operación.

4.3 CONEXIÓN CON AA_ENGINE

Cada jugador o NPC se conecta con el motor cuando se involucra la creación de una partida.

```
if usr != "" and passwd != "":
    client = MySocket("TCP", ADDR_E)
    client.send_msg("Player")
    client.send_obj((usr,passwd))
    msg = client.recv_msg()
    printMsg(msg)
    if msg[0:5] != "Error":
        while ready != "":
            ready = input("Presiona ENTER cuando esté todo listo: ")
        client.send_msg("Ready")
        mapa = getReady(usr)
        startMatch(usr, mapa)
```

Al conectarse al puerto *ADDR_E* se envía un mensaje indicando qué tipo de jugador se está conectando a la partida. Si es humano, se envía "Player" pero desde *AA_NPC* se envía como estado "NPC" para que la clase *Game* no le genere campos adicionales propios de humanos, como el movimiento por teclas.

Después se envía el usuario y la contraseña como objeto para saber, entre otras cosas, si el usuario existe o si ya está en la partida usando otro dispositivo. Si no recibe un mensaje de error, el cliente estará a la espera, donde si presiona la tecla, el servidor asumirá que está listo para empezar, y una vez todos los usuarios estén listos, se envíe el mapa por conexión.

```
def startMatch(alias, mapa):
    Thread(target=funcrecv, args=(mapa,alias,), daemon=True).start()
    t2 = Thread(target=funcsend, args=(alias,), daemon=True)
    t2.start()
    t2.join()
```

Como los jugadores son productores y consumidores simultáneamente, se ha decidido aplicar un par de hilos, uno que genere los mapas recibidos y otro que permita enviar al servidor, para ejecutar ambos bloques de código simultáneamente. Además, usando la función *join()* nos aseguramos de que salga del módulo cuando acabe de ejecutarse el hilo *t2*.

5. DESPLIEGUE

Al haber realizado la práctica en un lenguaje interpretado de alto nivel y usar pocas dependencias, no habría muchas complicaciones a la hora de desplegar el juego.

Sí que es necesario instalar *Python* (lenguaje base), *Kafka* y el módulo *Python-Kafka*. Y ya sería solamente ejecutar cada módulo como:

» Python <AA_Name> <args>

No obstante, y como ya se ha comentado, por simular un entorno real y aprender técnicas de despliegue se ha usado Docker para los módulos servidores.

Además, para cada dockerfile se ha creado un script en bash para automatizar la creación del contenedor y su ejecución con parámetros, de tal forma que para desplegar la práctica en un entorno Linux resultaría en:

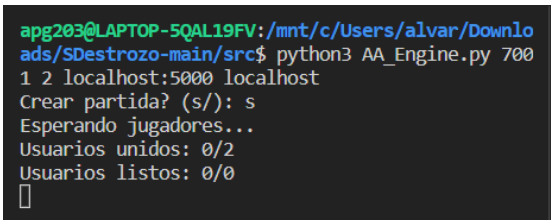
sh runa-prepare.sh
sh run-<module>.sh

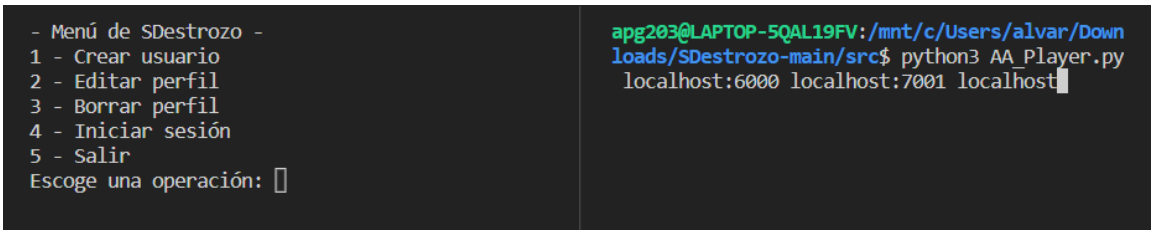
Caben destacar dos complicaciones al crear los dockerfiles:

No se puede acceder a un archivo fuera de la ruta donde se encuentre el dockerfile, por lo que en el runa-prepare.sh se copia los directorios de trabajo src y deploy a una carpeta release dentro del deploy con los dockers.

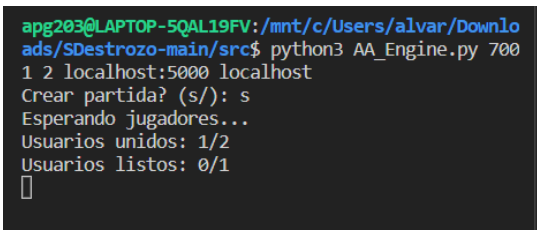
La base de datos no soporta conexiones, por lo que dos dockers no pueden compartir el mismo archivo SQLite. Por suerte existe opción de crear un volumen Docker para la base de datos y así poder compartirla entre contenedores. Como los dos servicios que la utilizan no lo hacen paralelamente no habría problemas con SQLite.

A continuación se presenta una simulación de lo que sería una partida real en un único dispositivo con solamente dos jugadores.

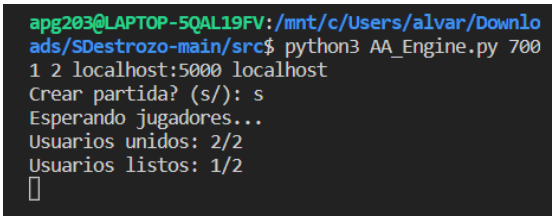




Cuando inicias el programa usando todos los parámetros correctamente, se abre el menú, donde puedes hacer las operaciones CRUD o jugar una partida iniciando sesión.



Al darle a unirse se puede apreciar como del lado del servidor se ha actualizado la cantidad de jugadores unidos pero no la de listos (hasta que no se presione ENTER no).



Al presionar ENTER, se actualiza la cantidad de jugadores listos, y una vez que todos los jugadores unidos estén listos, el servidor envía el mapa inicial. Los controles son:

Tecla	W	S	A	D	E	Q	C	X
Dirección	N	S	W	E	NE	NW	SE	SW

En este ejemplo de mapa, moveremos al jugador de la derecha (test) hacia la dirección E pulsando ‘d’.

Se puede apreciar que *test* se ha movido hacia el este, comiéndose un alimento por el camino, y que el mapa también se ha transmitido al usuario *aa*.

6. MEJORAS Y COMENTARIOS

Para la siguiente entrega se espera poder coger las temperaturas a tiempo real usando alguna especie de API como apoyo.

Con el fin de brindar más seguridad a la aplicación, se podría aplicar un hash a Kafka para codificar el alias de cada usuario.

Ahora mismo, tal y como está la entrega, la práctica es vulnerable a las caídas, algo que debemos mirar para la entrega final.

En cuanto al funcionamiento técnico, aún quedan algunos fallos menores por pulir pero se esperan solventar haciendo debugs, entre otras medidas.

En cuanto al progreso de la práctica, ha sido muy desnivelado. Al comienzo del curso se ha podido llevar un buen ritmo, pero por problemas con pygame, además de tener que rechazar la interfaz, retrasó considerablemente el resto de elementos técnicos, lo que ha derivado en que Javier haya hecho una gran parte del código existente y que esenciales como las acciones frente a las caídas no den tiempo suficiente para implementarse.

7. BIBLIOGRAFÍA

https://docs.python.org/3/howto/sockets.html

https://wiki.python.org/moin/UdpCommunication

https://docs.python.org/3/library/asyncio-stream.html

https://pythontic.com/modules/socket/bind

https://pythontic.com/modules/socket/connect

https://docs.python.org/3/library/struct.html

https://www.ibm.com/docs/es/oala/1.3.5?topic=SSPFMY_1.3.5/com.ibm.scala.doc/config/iwa_cnf_scldc_apche_con_c.htm

https://www.baeldung.com/ops/kafka-docker-setup