

SISTEMAS DISTRIBUIDOS

The title 'SISTEMAS DISTRIBUIDOS' is rendered in a bold, black, sans-serif font. The text is set against a vibrant, multi-colored background that resembles a rainbow or a spectrum of light. The colors transition from blue on the left, through purple, magenta, red, orange, and finally to yellow on the right. The background elements are curved and layered, creating a sense of depth and movement.

Samuel D. Bolanowski Ferrer
29525427M

Antonio Gómez Martínez
48791884Y

Índice

Tecnologías empleadas	1
AA_Engine	2
AA_Registry	4
Despliegue	4
Seguridad	5

Tecnologías empleadas

Antes de empezar con la explicación de nuestra implementación, vamos a explicar las tecnologías usadas en esta práctica.

Se ha utilizado React JS, un framework que permite usar lo que se conoce como “programación modular”, es decir, nos proporciona una gran cantidad de módulos para integrar lo necesario en nuestra aplicación. Además de esto, javascript es un lenguaje que ya conocíamos y entendíamos, por lo que lo hemos visto como la mejor opción.

Algunos de los paquetes que usamos en ReactJS son los siguientes: mongoose, bcryptjs, express, cors, socket.io, http, nodemon, aes-encryption.

Para ejecutar el módulo Front y Kafka hemos encontrado que la solución más fácil y viable ha sido utilizar AWS Lightsail, ya que este es un proveedor de servidor virtual privado (VPS) y es la manera más sencilla de conservar aplicaciones en la nube. Esto se ha realizado porque Antonio sabía de antemano que hacerlo en local daría problemas con la red de la Universidad.

AA_Engine

Respecto a la práctica anterior, incluye los siguientes cambios:

OpenWeather:

Desde el módulo de engine se hace una petición a la API de OpenWeather.

En el archivo **.env** se encuentra la clave para el acceso y uso de la API. El archivo **cities.txt** contiene un repertorio de ciudades que el usuario puede modificar a su gusto. De este conjunto el módulo elegirá aleatoriamente 4 ciudades para cada zona del tablero.

```
//Get rand Cities
const randomCities = () => {
  for (let i = CITY_ARR.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1));
    const aux = CITY_ARR[i];
    CITY_ARR[i] = CITY_ARR[j];
    CITY_ARR[j] = aux;
  }
  ciudad = CITY_ARR.slice(0, 4);
}
```

Después consultamos a API de OpenWeather la temperatura actual de estas 4 ciudades y guardamos la temperatura en un array de temperaturas.

```
var cityInfo = await
(axios.get("https://api.openweathermap.org/data/2.5/weather?q="
ciudad[i] + "&appid=" + key + "&units=metric"));
temps[i] = cityInfo.data.main.temp;
```

Seguridad:

AES es un algoritmo de cifrado simétrico que se utiliza para proteger la confidencialidad de los datos al transmitirlos a través de una red o almacenarlos de forma segura. La clave secreta es una contraseña que se utiliza para cifrar y descifrar los datos.

aes_encrypt: es una función que se utiliza para cifrar datos mediante el algoritmo de cifrado AES (Advanced Encryption Standard). AES es un

Práctica 3: Against All

algoritmo de cifrado simétrico que se utiliza para proteger la confidencialidad de la información.

```
/*  
=====   
==          Encryption          ==  
=====   
*/  
const AesEncryption = require('aes-encryption');  
const aes = new AesEncryption();  
const secreto = process.env.SECRET_AES;  
aes.setSecretKey(secreto);
```

Este trozo de código importa el módulo **aes-encryption** y crea una instancia de la clase **AesEncryption**. Luego, se establece una clave secreta para usar con el objeto **aes** utilizando la variable **secreto**, que se obtiene del entorno (**process.env.SECRET_AES**). Debe mantenerse en secreto y no compartirse con nadie más.

El orden de encriptado y desencriptado es el siguiente:

- Encriptación del payload.
- Envío del payload por un productor.
- Recepción del payload por un consumidor.
- Desencriptado del payload.

```
payload_PLAYER2MAP[0].messages = aes.encrypt(JSON.stringify(data));
```

```
weather = JSON.parse(aes.decrypt(message.value));
```

AA_Registry

Se ha creado una API que se encarga de registrar, iniciar sesión, y actualizar un usuario cuando la conexión por sockets falla. Esta se compone de ciertos modos que son característicos de las APIs:

- app.post('/login'): Manda datos para loggear.
- app.post('/register'): Manda datos para registrar el user.
- app.put('/update/user'): Manda datos para actualizar un user.

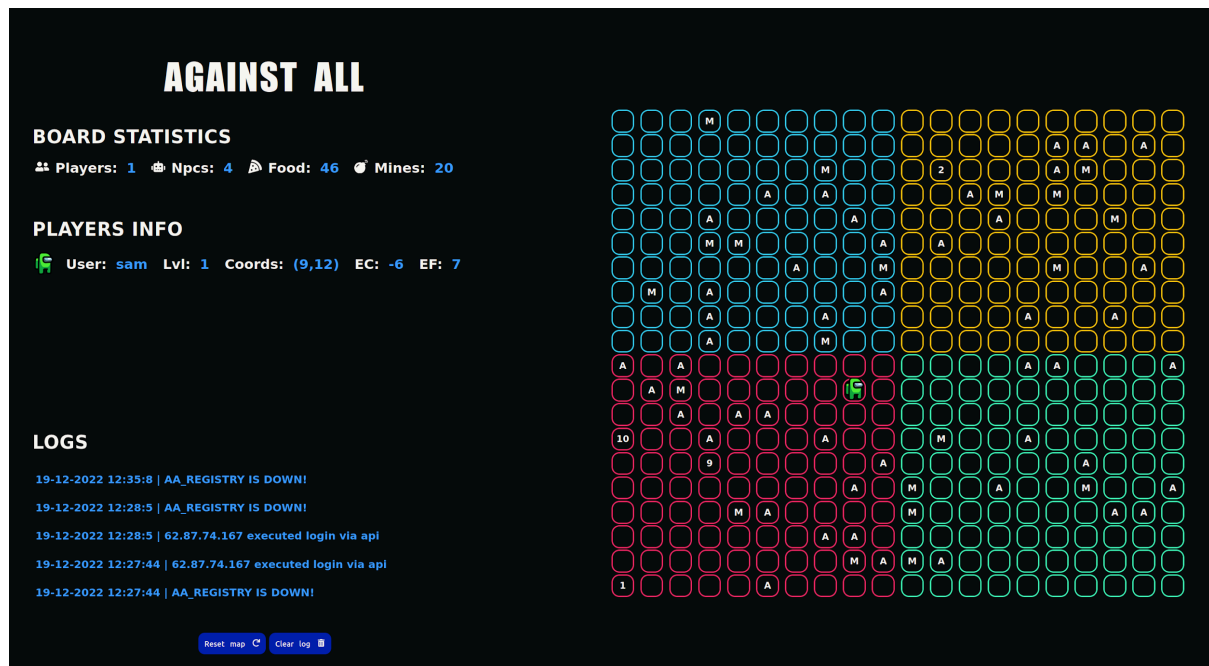
Esta parte ha sido muy sencilla, ya que solo se ha tenido que conectar a la base de datos y crear una API muy sencilla. En esta se ha implementado además la encriptación de la contraseña por medio del módulo bcryptjs que saltea la contraseña en función de los parametros dados.

```
> _id: ObjectId("633c4d087571a63f97021eee")
  user: "antuan"
 password: "$2a$12$eTyZbHY61R2LD0g2AF8AcudSqzXYiJqCa53Fjgu5IdRYiSzS01eBq"
  x: 0
  y: 0
  nvl: 1
  ec: 0
  ef: 0
  __v: 0
  img: "/img/pic_5.png"
```

AA_Front

Siguiente el diseño del módulo de player, se ha implementado una interfaz sencilla donde se puede visualizar distintos datos de la partida que se obtienen por medio de una API. También se dispone de una sección donde se pueden visualizar diferentes logs, donde puede aparecer por ejemplo cuando se cae un servicio.

Además, se han incorporado dos botones, uno para reiniciar el juego y otro para limpiar los logs.



Al front se puede acceder desde el siguiente enlace, cuando el VPS está activo y ejecutando la aplicación de React.

<http://antsam.online>

Para el dominio en el que hosteamos el front hemos utilizado AWS lightsail, este proporciona una muy buena opción para esta práctica. Algunos problemas que nos ha ocasionado ha sido que a la hora de poner el puerto nos redirige al 80 porque es el predeterminado, pero hemos conseguido que desde ese puerto nos muestre lo que hay en el puerto 3000, que es el que se usa en Reactjs.

Despliegue

Todos los módulos desarrollados cuentan con un archivo de configuración denominado **.env**, donde puedes declarar tus parámetros de la siguiente forma:

EJEMPLO_URI="holamundo.es/ejemplo"

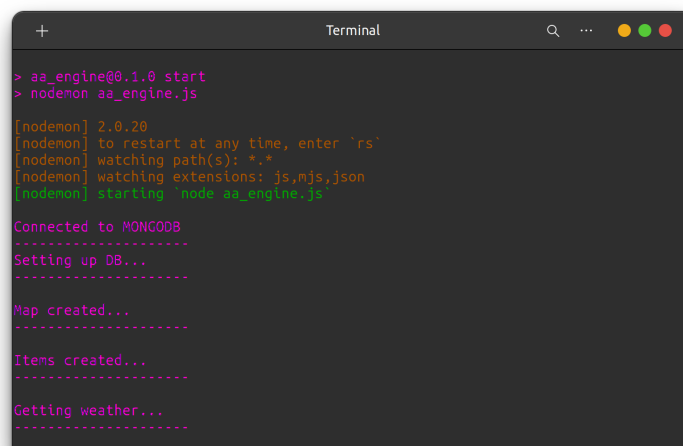
A continuación se detalla el orden en el que han de lanzarse los servicios para un funcionamiento óptimo y los parámetros disponibles.

Kafka: en nuestro VPS, se encuentra la imagen de docker, al ejecutar el comando **docker-compose up** inicializa kafka junto a zookeeper en la ip pública estática de nuestro servidor en el puerto 9092, previamente abierto.

AA_Front: se encuentra también en el VPS de Amazon Lightsail.

Tanto Kafka como el módulo AA_Front se inicializan por ssh con el script **./run_kafka.sh** y **./run_front.sh**. Para finalizar el servicio, con el atajo **CTRL + C** se detiene el módulo del front y con el script **./stop_kafka.sh** se detiene Kafka de forma correcta.

AA_Engine: este módulo en su archivo de configuración **.env** recibe las ip de Kafka, la key de OpenWeather, el URI a la base de datos MongoDB, y parámetros como el número de minas, de alimentos, y máximo de jugadores. Para ejecutarlo primero hay que ejecutar el instalador de paquetes **npm i** y a continuación **npm run start**.



```
+ Terminal
> aa_engine@0.1.0 start
> nodemon aa_engine.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting node aa_engine.js

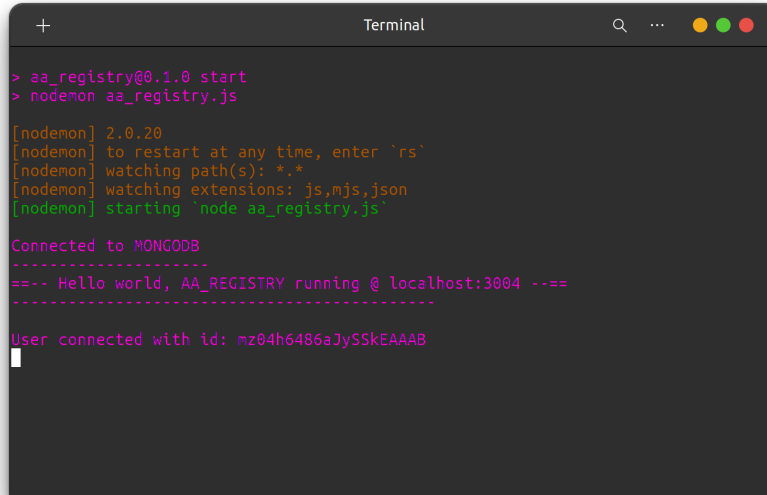
Connected to MONGODB
.....
Setting up DB...
.....

Map created...
.....

Items created...
.....

Getting weather...
.....
```


AA_Registry: en su archivo `.env` contiene los parámetros de los puertos destinados a comunicación por socket y api, la ip a utilizar y la URI de MongoDB. Para ejecutarlo primero hay que ejecutar el instalador de paquetes **npm i** y a continuación **./run.sh** para ejecutar registry por socket y api.



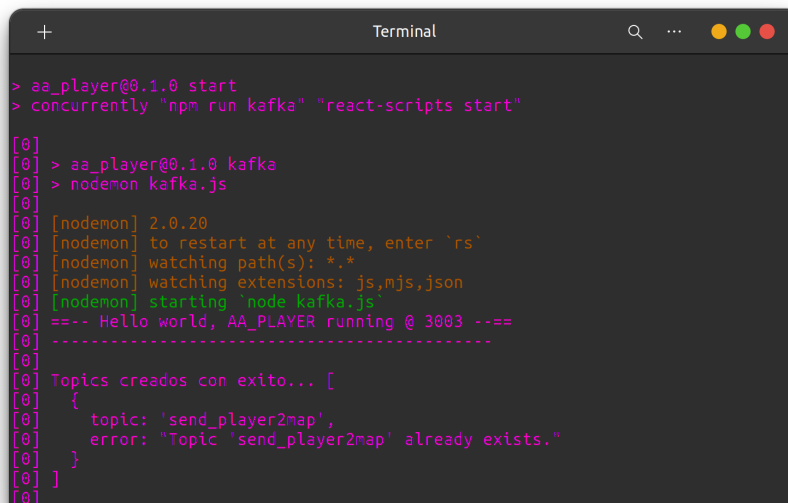
```
+ Terminal
> aa_registry@0.1.0 start
> nodemon aa_registry.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node aa_registry.js'

Connected to MONGODB
-----
=== Hello world, AA_REGISTRY running @ localhost:3004 ===
-----

User connected with id: mz04h6486aJySSkEAAAB
```

AA_Player: al contar con dos submódulos tiene dos archivos de configuración, para el controlador kafka el archivo `.env` contiene la ip del servicio Kafka, y en **src/config.json** el puerto y dirección al socket de AA_Registry. Para ejecutarlo primero hay que ejecutar el instalador de paquetes **npm i** y a continuación **npm run start**.



```
+ Terminal
> aa_player@0.1.0 start
> concurrently "npm run kafka" "react-scripts start"

[0] > aa_player@0.1.0 kafka
[0] > nodemon kafka.js
[0] [nodemon] 2.0.20
[0] [nodemon] to restart at any time, enter `rs`
[0] [nodemon] watching path(s): *.*
[0] [nodemon] watching extensions: js,mjs,json
[0] [nodemon] starting 'node kafka.js'
[0] === Hello world, AA_PLAYER running @ 3003 ===
[0] -----
[0] Topics creados con éxito... [
[0]   {
[0]     topic: 'send_player2map',
[0]     error: "Topic 'send_player2map' already exists."
[0]   }
[0] ]
```

Práctica 3: Against All

AA_NPC: su archivo de configuración `.env` tiene como parámetros la ip del servicio Kafka, la URI de MongoDB, y el número de npcs a crear. Para ejecutarlo primero hay que ejecutar el instalador de paquetes **npm i** y a continuación **npm run start**.

```

+ Terminal
> aa_npc@0.1.0 start
> nodemon aa_npc.js

[nodemon] 2.0.20
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting node aa_npc.js

Connected to MONGODB
-----
NPC created...
-----

Topics creados con éxito... [
  {
    topic: 'send_npc2map',
    error: "Topic 'send_npc2map' already exists."
  }
]

NPC enviado a mapa...
Sending payload result: { send_npc2map: { '0': 0 } }

```

Para las pruebas en laboratorio hay un script llamado `./run_test.sh` que ejecuta los dos módulos anteriores.

Ejemplo ilustrativo de cómo luce el front-end en una partida.

