



UNIVERSIDAD  
DE GRANADA

TRABAJO FIN DE GRADO  
INGENIERÍA INFORMÁTICA

# Desarrollo y evaluación de videojuego de tiempo real en línea

---

**Autor**  
David Erena Casado

**Directores**  
Juan José Ramos Muñoz



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

Granada, Septiembre de 2021







# **Desarrollo y evaluación de videojuego de tiempo real en línea**

---

**Autor**

David Erena Casado

**Directores**

Juan José Ramos Muñoz



# Desarrollo y evaluación de videojuego de tiempo real en línea

David Erena Casado

**Palabras clave:** online, Unity, Mirror, red, multijugador, videojuego

## Resumen

El sector de los videojuegos *online* sigue siendo un mercado interesante actualmente, con un número de usuarios cada vez mayor.

Sin embargo, esa gran cantidad de jugadores requiere resolver problemas de escalabilidad del servidor usado para el soporte de red, para cumplir con la demanda de los usuarios.

En este proyecto se ha diseñado, implementado y evaluado un juego multijugador online de acción 2D, donde los jugadores controlan a magos que lanzan hechizos generados de forma modular. Con el fin de que sea escalable, se ha diseñado una estructura basada en tres tipos de servidores que se comunican entre ellos, lo que permite lanzar servidores en diferentes máquinas.

Para realizarlo, se ha usado un motor de videojuegos ampliamente utilizado en la industria. Además, se han utilizado, tras comparar las opciones disponibles, bibliotecas y extensiones para ofrecer soporte de red y almacenamiento para almacenar los datos de los jugadores, diseñando una interfaz de programación de aplicación (API) para ello. También se desplegaron los servidores del juego mediante una máquina virtual en la nube de Amazon.

Finalmente, se ha medido el tráfico de red generado según la frecuencia de sincronización del servidor con los clientes, determinando cuál sería su valor adecuado para reducir ancho de banda sin menoscabar la calidad percibida por los jugadores, y se ha comprobado el correcto funcionamiento de los servidores distribuidos usando varias máquinas virtuales.



# **Development and evaluation of real-time multiplayer videogame**

David Erena

**Keywords:** online, Unity, Mirror, network, multiplayer, videogame

## **Abstract**

The online videogame industry is, to this day, an interesting market, with its number of players increasing daily.

However, that quantity of players requires solving problems related to the scalability of the server used for network support, in order to meet user demand.

In this project, an online multiplayer 2D action videogame was designed, implemented and evaluated. In this game, players take control of wizards that shoot modularly generated spells. To achieve scalability, a structure based on three different types of server was used, which allows to launch servers on different machines.

In order to carry out the project, a vastly popular videogame engine was chosen to be used. Furthermore, after comparing various options, other libraries and extensions have been utilised to support network play and keep track of player data, implementing an application programming interface (API) to compliment the latter. A virtual machine has been launched in Amazon's cloud to host the game's servers.

Lastly, network traffic has been measured, with relation to the frequency of synchronization from the server to the clients, choosing the right value to minimize bandwidth without taking away from the quality perceived by the players. Besides, the correct performance of servers distributed across different virtual machines has been tested.



---

Yo, **David Erena Casado**, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 26055001A, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: David Erena Casado

Granada a 6 de septiembre de 2021 .



---

D. **Juan José Ramos Muñoz**, Profesor del Área de Ingeniería Telemática del Departamento de Teoría de la Señal, Telemática y Comunicaciones de la Universidad de Granada.

**Informa:**

Que el presente trabajo, titulado ***Desarrollo y evaluación de videojuego de tiempo real en línea***, ha sido realizado bajo su supervisión por **David Erena Casado**, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 6 de Septiembre de 2021 .

**Los directores:**

**Juan José Ramos Muñoz**



# **Agradecimientos**

A los videojuegos. Son una cosa maravillosa...Es importante saber de qué van, pero lo más importante es decidirse a jugar a ellos.



# Índice general

<b>1. Introducción</b>	<b>21</b>
1.1. Objetivos del proyecto . . . . .	21
1.2. Organización de la memoria . . . . .	21
<b>2. Antecedentes y estado del arte</b>	<b>23</b>
2.1. Mercado de juegos de acción en tercera persona . . . . .	23
2.1.1. Lost Magic . . . . .	23
2.1.2. Nuclear Throne . . . . .	23
2.1.3. Duck Game . . . . .	24
2.2. Motores de videojuegos . . . . .	24
2.2.1. Unity . . . . .	24
2.2.2. GameMaker: Studio 2 . . . . .	25
2.2.3. Unreal Engine 4 . . . . .	26
2.2.4. Conclusión . . . . .	26
2.3. Frameworks de red para Unity . . . . .	27
2.3.1. UNet . . . . .	27
2.3.2. Photon Unity Networking (PUN) . . . . .	28
2.3.3. Mirror . . . . .	28
2.3.4. Conclusión . . . . .	29
<b>3. Especificación de requisitos</b>	<b>31</b>
3.1. Requisitos funcionales . . . . .	31
3.2. Requisitos no funcionales . . . . .	32
<b>4. Planificación</b>	<b>35</b>
4.1. Metodología . . . . .	35
4.1.1. Cómo funciona SCRUM . . . . .	35
4.1.2. Adaptación de SCRUM a un equipo de una persona .	36
4.2. Estimación temporal . . . . .	36
4.2.1. Diagrama de Gantt . . . . .	38
4.3. Estimación de costes . . . . .	45
4.4. Desviación de las estimación . . . . .	46

<b>5. Diseño</b>	<b>49</b>
5.1. Diseño del juego . . . . .	49
5.1.1. Descripción del juego . . . . .	49
5.2. Descripción global del diseño de juego . . . . .	49
5.2.1. Servidor de matchmaking . . . . .	51
5.2.2. Servidor de partida . . . . .	51
5.2.3. Servidor de combate . . . . .	53
5.2.4. Cliente . . . . .	54
5.3. Algoritmo de ataque . . . . .	54
5.4. Comunicación entre las entidades . . . . .	58
5.4.1. Actualización de posición de instancias . . . . .	58
5.4.2. Mensajes . . . . .	58
5.4.3. Funcionalidades . . . . .	60
5.5. Interfaz . . . . .	70
5.5.1. Diagrama de flujo entre pantallas . . . . .	70
5.5.2. Interfaz de menú . . . . .	72
5.5.3. Interfaz de combate . . . . .	75
5.5.4. Interfaz de partida . . . . .	77
<b>6. Implementación</b>	<b>79</b>
6.1. Base de datos de jugadores . . . . .	79
6.1.1. Base de datos . . . . .	79
6.1.2. Comunicación con MongoDB . . . . .	79
6.1.3. Consistencia de datos . . . . .	79
6.2. Identificación de los jugadores . . . . .	80
6.3. Creación de servidores . . . . .	80
6.3.1. Script de creación . . . . .	80
6.4. Esperas dentro de un proceso . . . . .	81
6.5. Interfaz de usuario . . . . .	81
6.6. Combate . . . . .	82
6.6.1. Hechizos . . . . .	82
6.6.2. Instanciamiento de jugadores . . . . .	82
6.6.3. Sincronización entre jugadores . . . . .	83
6.7. Comunicación entre las entidades . . . . .	83
6.7.1. Tipo de transporte . . . . .	83
6.7.2. Mensajes . . . . .	83
6.8. Despliegue de la aplicación en Amazon Web Services . . . . .	84
6.8.1. Configuración de máquina virtual . . . . .	84
6.8.2. Despliegue del proyecto . . . . .	84
<b>7. Resultados experimentales</b>	<b>87</b>
7.1. Prueba sobre la tasa de refresco de componentes de sincronización . . . . .	87
7.1.1. Entorno experimental . . . . .	87

7.1.2. Definición de la prueba . . . . .	88
7.1.3. Parámetros de la prueba . . . . .	89
7.1.4. Resultados experimentales . . . . .	90
7.2. Prueba de instanciación remota de servidores para escalado . .	95
7.2.1. Entorno . . . . .	96
7.2.2. Definición de la prueba . . . . .	96
7.2.3. Parámetros de la prueba . . . . .	96
7.2.4. Resultados experimentales . . . . .	96
<b>8. Conclusiones</b>	<b>101</b>
8.1. Conclusiones . . . . .	101
8.2. Trabajo futuro . . . . .	101
<b>A. Instrucciones de uso</b>	<b>103</b>
<b>Glosario</b>	<b>109</b>



# Índice de figuras

4.1. Diagrama de Gantt de la planificación del proyecto . . . . .	38
4.2. Diagrama de Gantt de la planificación del sprint 1 . . . . .	39
4.3. Diagrama de Gantt de la planificación del sprint 2 . . . . .	40
4.4. Diagrama de Gantt de la planificación del sprint 3 . . . . .	41
4.5. Diagrama de Gantt de la planificación del sprint 4 . . . . .	41
4.6. Diagrama de Gantt de la planificación del sprint 5 . . . . .	42
4.7. Diagrama de Gantt de la planificación del sprint 6 . . . . .	42
4.8. Diagrama de Gantt de la planificación del sprint 7 . . . . .	43
4.9. Diagrama de Gantt de la planificación del sprint 8 . . . . .	44
5.1. Diagrama de diseño . . . . .	50
5.2. Diagrama de clase del servidor de matchmaking . . . . .	52
5.3. Diagrama de clase del servidor de partida . . . . .	53
5.4. Diagrama de clase del servidor de combate (Primera mitad) .	54
5.5. Diagrama de clase del servidor de combate (Segunda mitad) .	55
5.6. Diagrama de clase del cliente (Primera mitad) . . . . .	56
5.7. Diagrama de clase del cliente (Segunda mitad) . . . . .	57
5.8. Diagrama de secuencia del proceso de Login . . . . .	61
5.9. Diagrama de secuencia del proceso de registro . . . . .	62
5.10. Diagrama de secuencia para validar la presencia de un jugador en servidor de partida . . . . .	63
5.11. Diagrama de secuencia para validar la presencia de un jugador en servidor de combate . . . . .	65
5.12. Diagrama de secuencia sobre cambiar conexión . . . . .	66
5.13. Diagrama de secuencia sobre informar de resultado de combate	66
5.14. Diagrama de secuencia sobre informar de resultado de partida	67
5.15. Diagrama de secuencia sobre asignar color a un jugador . .	68
5.16. Diagrama de secuencia sobre buscar partida . . . . .	69
5.17. Diagrama de secuencia sobre cerrar sesión . . . . .	70
5.18. Diagrama de flujo de las interfaces . . . . .	71
5.19. Boceto de interfaz de menú . . . . .	72
5.20. Boceto de interfaz de login . . . . .	73
5.21. Boceto de interfaz de registro . . . . .	74

5.22. Boceto de interfaz de pre-combate . . . . .	75
5.23. Boceto de interfaz de combate . . . . .	76
5.24. Boceto de interfaz entre rondas . . . . .	77
5.25. Boceto de interfaz de partida finalizada . . . . .	78
6.1. Ejemplo de reglas de seguridad del firewall de AWS . . . . .	84
7.1. Especificaciones de la máquina usada . . . . .	88
7.2. Ejemplo de uso de Wireshark . . . . .	89
7.3. Gráfica de cambio medio en la x del jugador con respecto al intervalo de sincronización . . . . .	91
7.4. Gráfica de media de bitrate (Cliente a servidor) . . . . .	92
7.5. Gráfica de paquetes generados por segundo (Cliente a servidor) . . . . .	93
7.6. Gráfica de media de bitrate generado (Servidor a cliente) . . . . .	94
7.7. Gráfica de paquetes generados por segundo (Servidor a cliente) . . . . .	95
7.8. Gráfica de comparación de la carga CPU lanzando todos los servidores en local. . . . .	97
7.9. Gráfica de carga de memoria en local . . . . .	98
A.1. Pantalla de menú . . . . .	104
A.2. Pantalla de partida . . . . .	105
A.3. Pantalla previa al combate . . . . .	106
A.4. Pantalla de combate . . . . .	107

# Índice de cuadros

2.1. Tabla resumen de características de motores . . . . .	27
2.2. Tabla resumen de características de frameworks multijugador para Unity . . . . .	29
4.1. Tabla de presupuesto de hardware del proyecto . . . . .	45
4.2. Presupuesto software del proyecto . . . . .	46
4.3. Presupuesto de salario . . . . .	46
4.4. Presupuesto total resumido del proyecto . . . . .	46
7.1. Tabla de resultados de cambio de movimiento en x. . . . .	90
7.2. Tabla de tráfico generado desde el cliente al servidor. . . . .	91
7.3. Tabla de paquetes generados (Cliente a servidor). . . . .	92
7.4. Tabla de tráfico generado (Servidor a cliente). . . . .	93
7.5. Tabla de paquetes generados desde el servidor al cliente. . . . .	94
7.6. Tabla de carga de CPU lanzando todo en local . . . . .	97
7.7. Tabla de carga de memoria lanzando todo en local . . . . .	97
7.8. Tabla de carga de memoria lanzando servidores en máquinas virtuales . . . . .	99



# Capítulo 1

## Introducción

La industria de los videojuegos se encuentra en auge, no solo de forma internacional sino también de forma nacional. El sector productor de videojuegos españoles facturó en 2019 920 millones de euros, un 13 % más que 2018. Según las previsiones, la facturación crecerá a un ritmo anual del 17 %, lo que supondría superar en 2023 los 1.700 millones de euros [1].

Más específicamente, la industria de los videojuegos multijugador sigue siendo una gran fuente de ingresos [2], y sectores como el de los *e-sports* siguen creciendo [3].

Con este crecimiento en el que más jugadores se unen de forma masiva a juegos multijugador [4], surge el problema de lidiar con la escalabilidad que requiere dicha masa de jugadores, la cual normalmente se realiza desde un solo servidor. Dicho problema es el que se busca solucionar en este proyecto.

### 1.1. Objetivos del proyecto

En este proyecto se busca el desarrollo de un videojuego multijugador que pueda distribuir su carga entre varias entidades, de forma que este sea escalable sin afectar a la experiencia de sus jugadores.

### 1.2. Organización de la memoria

La memoria está estructurada en 8 capítulos donde se tratarán distintos aspectos del proyecto, de acuerdo a la siguiente distribución:

En el capítulo 2, se abordan las similitudes con otros juegos y se estudia qué motor a elegir para el proyecto.

En el capítulo 3, se abordan los requisitos tanto funcionales como no funcionales a cumplir en este proyecto.

En el capítulo 4, se aborda la planificación en metodología, tiempo y costes del proyecto.

En el capítulo 5, se aborda todo lo relevante al diseño del proyecto, tanto desde el punto de vista del juego como la comunicación entre servidores.

En el capítulo 6, se aborda la materialización de todo lo expuesto en el capítulo 5

En el capítulo 7, se aborda el rendimiento y tráfico de red que provoca el proyecto.

En el capítulo 8, se aborda las conclusiones obtenidas al realizar el proyecto y medidas a realizar como trabajo futuro.

## Capítulo 2

# Antecedentes y estado del arte

En este capítulo se explican las características en común tras las referencias del proyecto y se estudia qué herramientas se encuentran disponibles para llevarlo a cabo.

### 2.1. Mercado de juegos de acción en tercera persona

El proyecto a desarrollar es un juego de acción en tercera persona multi-jugador, haciendo uso de un sistema de hechizos creados de forma modular. En esta sección se analizan otros juegos del mismo género para comprobar características esenciales y aspectos a mejorar de cada uno.

#### 2.1.1. Lost Magic

Lost Magic es un juego RPG y de estrategia en tiempo real desarrollado por Taito para Nintendo DS [5]. En él se controla a un mago que puede lanzar una serie de hechizos trazándolos en la pantalla táctil, e incluso mezclar hechizos en grupos de hasta tres para crear otros más potentes.

Los hechizos se mezclan de forma modular, de forma que si se elige de primero el mismo para dos grupos distintos, se puede aproximar cómo serán los hechizos resultantes. Esta mecánica se usa en el proyecto, pero adaptada a controles de ratón y teclado y bajo el contexto de un juego de acción más frenético que Lost Magic.

#### 2.1.2. Nuclear Throne

Nuclear Throne es un juego *shoot 'em up roguelike* desarrollado por Vlambier, donde se controla a un personaje a través de un número de fases eli-

minando a enemigos y mejorando de nivel [6].

Es un referente para este juego, puesto que ambos son similares en el aspecto gráfico (perspectiva, aspecto *pixel art...*) y en la velocidad de juego.

### 2.1.3. Duck Game

Duck Game es un videojuego de acción desarrollado por Landon Podbielski. Se basa en combates muy frenéticos que mezclan disparos y plataformas [7]. Su modo multijugador consiste en varios combates donde al ganador se le otorga un punto, y gana la partida el primer jugador en llegar a 10 puntos.

En el proyecto se usa un sistema parecido a este, ya que los jugadores se enfrentan entre ellos durante un número de rondas específico para conseguir puntos y clasificarse para una final donde se enfrentarán los mejores clasificados.

## 2.2. Motores de videojuegos

Un motor de videojuegos es un framework de software dedicado a la creación y desarrollo de estos. [8]

Para este proyecto, se necesita un motor con soporte para multijugador online, y se valorarán otros factores como que sea accesible para desarrolladores con poca experiencia, de coste bajo y que permita trabajar de forma rápida. Existen varios que se ajustan a estas características, a continuación se describen los más destacados:

### 2.2.1. Unity

Unity es un motor de desarrollo multiplataforma desarrollado por Unity Technologies. Desde su anuncio en 2005, ha sido extendido para soportar distintas plataformas tanto en móvil como en consolas y PC. Es conocido por ser de fácil adaptación para desarrolladores recién empezados y muy usada en el desarrollo indie. Puede usarse tanto para videojuegos en 3D como en 2D, y además se usa para otras industrias aparte del videojuego, como cine, ingeniería y arquitectura.[9]

#### Ventajas

Unity tiene varias características que lo hacen ventajoso frente a motores de videojuegos clásicos:

- Flujo de trabajo ágil, permite prototipar rápido e ir lanzando resultados poco a poco, lo cual viene bien para desarrollo ágil, e incluso para poder realizar prototipos sobre cómo debe ser el producto final.

- Una tienda de contenidos integrada (Asset Store), llena de plug-ins y demás útiles a la hora de desarrollar, que además se puede acceder desde el mismo motor, agilizando el proceso de instalación de herramientas.
- Fácil de usar para principiantes. Gran comunidad llena de contenido a la hora de aprender.
- Ofrece facilidades en desarrollo 2D.
- Licencias gratuitas para desarrollos no profesionales.

### Desventajas

- El soporte multijugador de Unity está obsoleto, pero se puede paliar con frameworks de otras fuentes, como por ejemplo con Mirror [10] o Photon [11].
- Uso gratuito limitado: ciertas características, como la monetización, solo forman parte de planes de Unity pagados.

#### 2.2.2. GameMaker: Studio 2

Es una plataforma basada en un lenguaje de programación interpretado y un kit de desarrollo de software (SDK) para desarrollar videojuegos, creada originalmente por el profesor Mark Overmars en el lenguaje de programación Delphi, y orientada a usuarios novatos o con pocas nociones de programación [12].

### Ventajas

- Muy fácil de usar. Incluso con falta de conocimientos en programación ya que proporciona programación visual.
- La facilidad de uso proporciona una forma rápida de realizar prototipos.
- Proporciona más herramientas para desarrollo (edición de sprites, de audio, ...)
- Especializado en desarrollo 2D. Muchos videojuegos indies de éxito están desarrollados en Game Maker Studio. [13]

### Desventajas

- De pago, ya que gratuitamente no se pueden exportar proyectos.
- Poca ayuda de cara a desarrollar un juego multijugador, se realizaría la estructura desde cero.

### 2.2.3. Unreal Engine 4

Unreal Engine es un motor de juego creado por la compañía Epic Games, mostrado inicialmente en el shooter en primera persona Unreal en 1998. Presenta un alto grado de portabilidad y es una herramienta utilizada actualmente por muchos desarrolladores de juegos, sobre todo en el desarrollo de videojuegos AAA [14].

#### Ventajas

- Motor consolidado en la industria debido a su robustez ya probada.
- Es de código abierto, que permite a la comunidad hacer mejoras sobre el motor.
- Uso gratuito, Unreal consigue un 5 % de los beneficios si estos exceden el millón de dólares.
- El multijugador ofrecido por Unreal es más potente que el ofrecido por los otros dos motores de base.

#### Desventajas

- Aprendizaje más complicado que en los otros dos motores, lo que ralentizaría el desarrollo del proyecto.
- No permite un flujo de trabajo tan rápido como Unity o Game Maker Studio.
- Menos enfocado en el desarrollo 2D que el resto.

### 2.2.4. Conclusión

En primer lugar, no se considera como opción para el presente desarrollo el entorno de Game Maker Studio 2 por su escasa capacidad para multijugador, que puede ser determinante en el desarrollo del proyecto.

Aunque Unreal satisface los requisitos del proyecto, su soporte para gráficos 2D es menos claro que en otros motores, y la curva de aprendizaje puede ser más pronunciada.

Finalmente, se elige Unity ya que es un motor más enfocado hacia el desarrollo 2D, que permite trabajar y adaptarse de forma rápida, y no se pierde tanto en opciones sobre multijugador debido a los distintos frameworks de otros fabricantes.

Tabla resumen			
	Unity	Game Maker Studio 2	Unreal Engine 4
Accesibilidad	Media	Alta	Baja
Capacidad para multijugador	Amplia usando frameworks: Mirror, Photon, ...	Débil, hay poca ayuda, infraestructura desde cero	Potente
Precio	Gratis	8.19€/mes	Gratis
Enfoque 2D	Sí	Sí	No
Velocidad de trabajo	Rápida	Rápida	Lenta

Cuadro 2.1: Tabla resumen de características de motores

## 2.3. Frameworks de red para Unity

Existen varios frameworks para multijugador online de Unity. En esta sección se comparan los más destacados con la intención de elegir el más apropiado para el proyecto:

### 2.3.1. UNet

UNet es la solución multijugador de Unity [15], la cual está en plan de deprecación desde la versión 2018.4 (LTS) [16]. Ofrece la siguientes herramientas:

- **Remote Procedure Calls (RPC)** y **TargetRPCs**: Permiten llamar a una función desde el servidor en todos los clientes o solo en uno, respectivamente.
- **Commands**: Permiten, desde el cliente, ejecutar una función en el servidor.
- **NetworkTransform**: Componente de Unity que permite sincronizar el transform (Posición, rotación y escala) de un objeto en todos los clientes.
- **NetworkManager**: Permite gestionar las funciones referentes al cliente y servidor desde un mismo script, simplemente sobreescribiendo funciones de UNet (Por ejemplo: OnClientDisconnect, OnServerAddPlayer, ...)
- **UnityWebRequest**: Clase de Unity para realizar peticiones HTTP.
- La conexión puede ser tanto cliente/cliente como cliente/servidor.

Se descarta usar este framework debido a la falta de apoyo de Unity y su obsolescencia.

### **2.3.2. Photon Unity Networking (PUN)**

PUN es la solución de Photon para el multijugador de Unity [11]. Entre sus características principales se encuentran:

- Mayormente usado para experiencias multijugador de duración corta y con pocos jugadores
- Fácil de usar y conexión garantizada entre jugadores (usa el servidor de relay de Photon, con lo que la conectividad falla solo si falla el servidor).
- Trae una implementación de Matchmaking y migración de host por defecto.
- La conexión es cliente/cliente, con un cliente que alberga la partida.
- Gratis para 20 usuarios concurrentes, para más es necesario pagar.

En cuanto a herramientas, ofrece otras implementaciones de las ya vistas en UNet, como RPCs y sincronización de propiedades entre jugadores [17].

### **2.3.3. Mirror**

Mirror es un framework de multijugador que surge a partir del obsoleto UNet, y desarrollado por la comunidad ante la falta de respuesta de Unity sobre un nuevo sistema de multijugador [10]. Ofrece las mismas herramientas que UNet, pero con un soporte mucho mayor apoyado en la comunidad y una gran cantidad de bugs arreglados y mejoras. Por lo tanto, tiene las mismas ventajas que UNet, como la posibilidad de conexión servidor-cliente, y edición simple de objetos a través de componentes de Unity. Además, al igual que UNet, es completamente gratuito de usar.

### 2.3.4. Conclusión

En el cuadro 2.3.4 se realiza un resumen de las características de cada framework explicado anteriormente.

<b>Tabla resumen</b>			
	<b>UNet</b>	<b>Photon</b>	<b>Mirror</b>
Tipo de comunicación	Cliente/Servidor	Cliente/Cliente	Cliente/Servidor
Apoyo	Obsoleto	Menor que Mirror, pero ofrece muchas facilidades adicionales	Llevado por su comunidad y puede apoyarse en otros assets de Unity
Precio	Gratis	Gratis hasta 20 CCU	Gratis

Cuadro 2.2: Tabla resumen de características de frameworks multijugador para Unity

A pesar de las muchas facilidades que ofrece Photon, en este trabajo se valora mucho tanto la comunidad alrededor de Mirror como la posibilidad de comunicación cliente/servidor (que ayudará a evitar trampas) como que sea completamente gratuito, por lo que finalmente se adoptará Mirror.



## Capítulo 3

# Especificación de requisitos

Tal como se ve en la asignatura Fundamentos de Ingeniería de Software (FIS), haciendo uso de la bibliografía [18] y [19], se especifican los requisitos funcionales y no funcionales:

### 3.1. Requisitos funcionales

Citando a [18]: "Son enunciados acerca de servicios que el sistema debe proveer, de cómo debería reaccionar el sistema a entradas particulares y de cómo debería comportarse el sistema en situaciones específicas. En algunos casos, los requerimientos funcionales también explican lo que no debe hacer el sistema".

En este proyecto, los requisitos funcionales especificados han sido los siguientes:

**RF-1.** *Combate.* Los jugadores participarán en combates, cuyas mecánicas básicas serán las siguientes:

**RF-1.1.** *Movimiento.* Los jugadores se moverán en cualquier dirección por el campo de juego.

**RF-1.2.** *Cámara.* La cámara de cada jugador mostrará una visión de los alrededores del jugador visto desde arriba.

**RF-1.3.** *Disparo simple.* Los jugadores podrán disparar diferentes hechizos de forma individual.

**RF-1.4.** *Disparo combinado.* Los jugadores podrán disparar grupos de hasta 3 hechizos. El daño y velocidad de este hechizo será una combinación de los daños de los dos primeros hechizos mientras que el tercero le aplicará una función que lo cambiará de alguna forma.

**RF-1.5.** *Salud y maná.* El estado de los jugadores quedará determinado por su salud y maná restantes, que se actualizará al

recibir el impacto de un hechizo ajeno y al disparar un hechizo, respectivamente. En el caso del maná, este deberá regenerarse poco a poco.

**RF-2.1.** *Puntuación.* Tras cada combate, se otorgarán puntos a los jugadores según cuándo hayan sido derrotados (u ordenados por salud si no lo han sido).

**RF-2.** *Partida.* Los jugadores jugarán partidas:

**RF-2.1.** *Clasificación.* Habrá una tabla que indique la puntuación de cada jugador en orden. Esta tabla se deberá mostrar a los jugadores entre combates.

**RF-2.2.** *Rondas.* La partida se estructurará en rondas, habiendo en cada ronda combates entre jugadores emparejados aleatoriamente.

**RF-2.3.** *Combate final.* Tras jugarse un número determinado de rondas, se jugará un combate final entre los mejores clasificados hasta el momento, donde se decidirá el ganador de la partida.

**RF-2.4.** *Final de partida.* Al acabar la partida, cada jugador decidirá si quiere buscar partida de nuevo o cerrar sesión.

**RF-3.** *Gestión de cuentas.* Los jugadores deberán registrarse/loguearse para poder jugar.

**RF-3.1.** *Login.* Los jugadores deberán loguearse usando su nombre de usuario y contraseña para poder buscar partida.

**RF-3.2.** *Registrarse.* Los jugadores se registrarán con su nombre de usuario y su contraseña.

## 3.2. Requisitos no funcionales

Citando de nuevo a [18]: ”Son limitaciones sobre servicios o funciones que ofrece el sistema. Incluyen restricciones tanto de temporización y del proceso de desarrollo, como impuestas por los estándares. Los requerimientos no funcionales se suelen aplicar al sistema como un todo, más que a características o a servicios individuales del sistema”.

En este proyecto, los requisitos no funcionales encontrados han sido los siguientes:

**RNF-1.** *Seguridad en el login.* El login ha de ser lo más seguro posible para evitar que se filtren datos de inicio de sesión de los usuarios.

**RNF-2.** *Cambio rápido de hechizos.* El cambio entre hechizos ha de ser rápido ya que, en caso contrario, afectará al combate y será frustrante para los jugadores.

**RNF-3.** *Seguridad en contraseña.* Al registrarse, el jugador deberá escribir dos veces la contraseña.

**RNF-4.** *Evitar doble registro.* No se podrá registrar un jugador si ya existe otro con el mismo nombre de usuario.

**RNF-5.** *Información antes de combate.* Se mostrará a cada jugador los participantes en cada uno de sus combates.

**RNF-6.** *Jugadores diferenciados visualmente.* Cada jugador tendrá un color diferente al resto de jugadores.



# Capítulo 4

## Planificación

### 4.1. Metodología

La metodología escogida para el proyecto es SCRUM [20], una metodología de desarrollo ágil de uso muy generalizado y resultados probados.

#### 4.1.1. Cómo funciona SCRUM

De la Guía de Scrum [21]: "Scrum es un marco ligero que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptables para problemas complejos. En pocas palabras, Scrum requiere un Scrum Master para fomentar un entorno donde:

1. Un propietario del producto (Product Owner) ordena el trabajo de un problema complejo en un Product Backlog.
2. El equipo de Scrum convierte una selección del trabajo en un Incremento de valor durante un Sprint.
3. El equipo de Scrum y sus partes interesadas (stakeholders) inspeccionan los resultados y realizan los ajustes necesarios para el próximo Sprint.
4. Repetir." [21]

Dicho de otra forma, se basa en diferentes roles que se complementan: *Scrum Master*, *Product Owner* (propietario del producto) y *Development Team* (equipo de desarrollo). Todos conforman el *Scrum Team*.

El *Product Owner*, que es el interesado en el desarrollo de un producto, comunica al resto de partes su visión de este a través del *Product Backlog*, una pila que almacena todas las tareas a completar por el *Development Team*.

Para desarrollar el producto, el *Development Team* se organiza en *Sprints*, períodos de tiempo (sin variar la duración) donde se completan una serie

de tareas del *Product Backlog*, las cuales se organizan en el *Sprint Backlog*. Cada Sprint comienza con el *Sprint Planning*, donde el equipo se reúne para discutir el objetivo del Sprint y qué valor le otorga al producto, además de planificar el trabajo mediante la descomposición de elementos del *Product Backlog* en tareas más asequibles.

Además, cada día se realiza el denominado *Daily Scrum*, o Scrum Diario. Consiste en que los desarrolladores se reúnen durante 15 minutos máximo para planificar ese día de trabajo, aunque pueden reunirse más veces a lo largo del día para replanificar o adaptar el resto del Sprint.

Finalmente, a modo de conclusión del Sprint, se realizan el *Sprint Review* y *Sprint Retrospective*. El primero sirve a modo de inspección sobre lo que se ha conseguido en el Sprint y qué hacer a continuación, mientras que el segundo se fija en la productividad del equipo, las herramientas usadas, etc, con tal de hacer cambios que aporten a la eficacia del equipo.

En este proceso, el *Scrum Master* se encarga de que la metodología se lleve a cabo de la mejor forma posible. Para ello, elimina obstáculos del equipo de desarrollo, los ayuda a centrarse, y sirve de enlace entre estos y el *Product Owner*, facilitando la comunicación entre todas las partes.

#### 4.1.2. Adaptación de SCRUM a un equipo de una persona

Scrum está diseñado para equipos pequeños (10 personas o menos) [21] pero que todos los roles los lleve una sola persona supone un caso especial.

SCRUM para una sola persona supone cumplir todos los roles del Scrum Team, detallados en la sección anterior. De tal forma, es responsabilidad del individuo definir lo mejor posible el producto tal y como haría un Product Owner, resolver problemas y conseguir llevar a cabo la metodología lo mejor posible como haría un Scrum Master, y planificar sprints y realizar el desarrollo del producto en sí como Development Team.

Se usaron sprints de dos semanas, que otorga tiempo para implementar funciones de peso en cada uno. A la hora de organizar las tareas a realizar en cada uno, se usó la plataforma Trello, que permite organizarlas en distintas tarjetas y tablas según el nivel de completitud de cada una.

## 4.2. Estimación temporal

El proyecto se planificó en ocho sprints, los cuales, de forma resumida, tenían los siguientes objetivos:

- Sprint 1: Desarrollo de mecánicas básicas y UI.
- Sprint 2: Dinámica de juego de combate simple.
- Sprint 3: Estructura de servidores (Servidor de combate y partida).

- Sprint 4: Finalizar un combate.
- Sprint 5: Sistema de rondas y puntuación.
- Sprint 6: Servidor Matchmaking y comunicación.
- Sprint 7: MongoDB (Login/Registrar).
- Sprint 8: Despliegue en Amazon Web Services (AWS).

A continuación se detalla más la planificación del proyecto y se profundiza más en las tareas que se estimaron para cada sprint.

#### 4.2.1. Diagrama de Gantt

En la figura 4.1 se detalla la planificación temporal del proyecto:

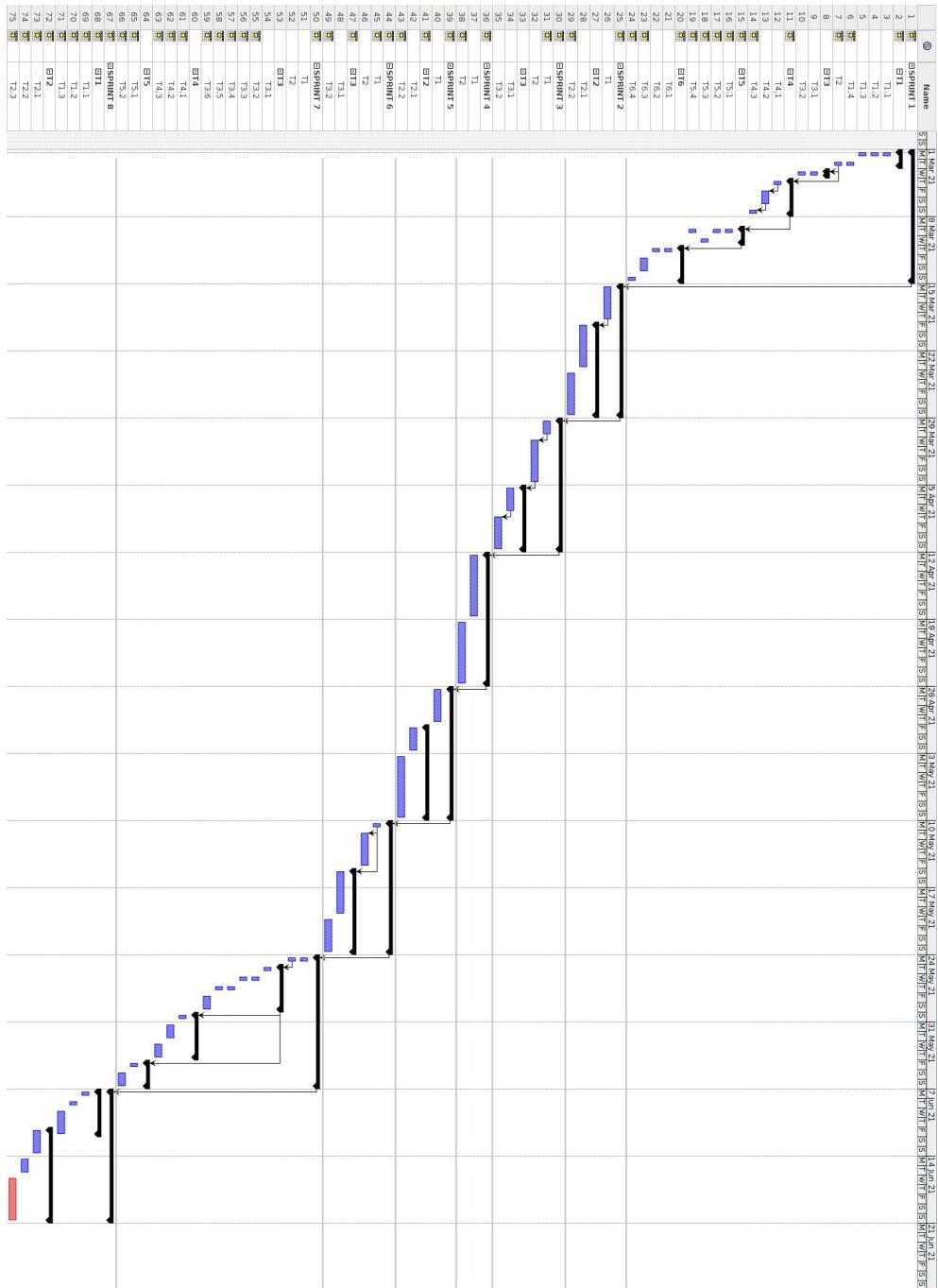


Figura 4.1: Diagrama de Gantt de la planificación del proyecto

### SPRINT 1. Desarrollo de mecánicas básicas y UI.

Se detalla el planteamiento del sprint mediante un diagrama de Gantt en la figura 4.2 y posteriormente las tareas que lo conforman:

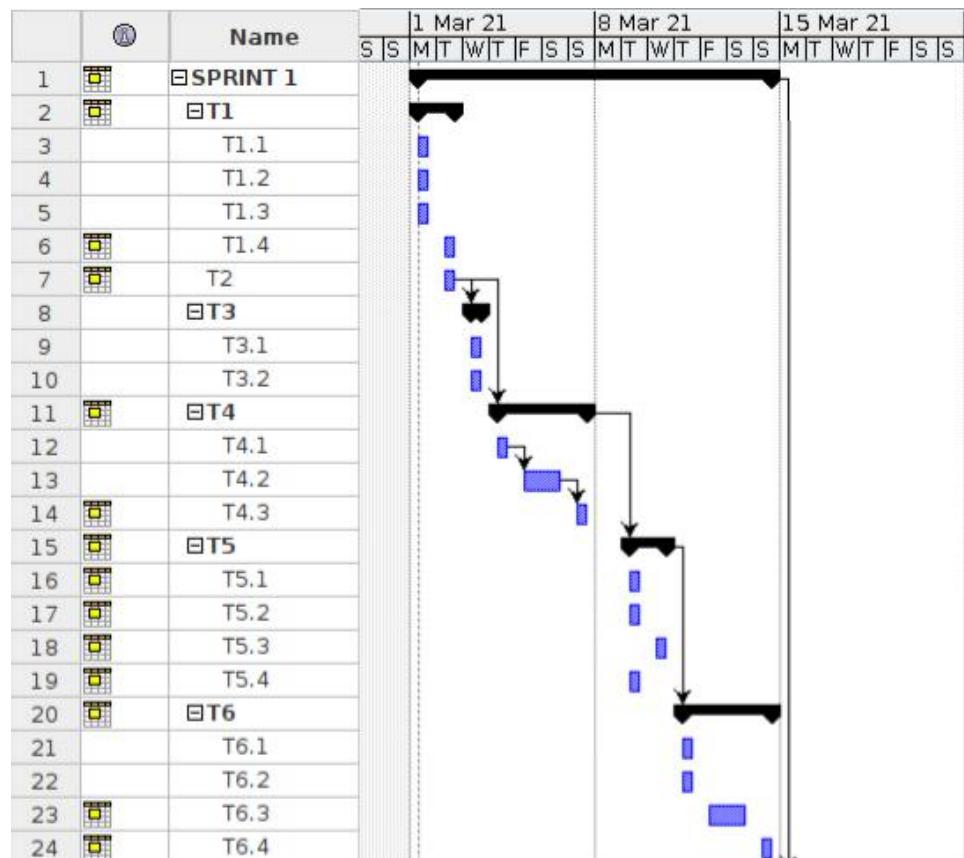


Figura 4.2: Diagrama de Gantt de la planificación del sprint 1

**T1.** Movimiento.

**T1.1.** Crear objeto jugador.

**T1.2.** Mandar command a servidor de combate.

**T1.3.** Ejecutar RPC de sincronización de posición.

**T1.4.** Implementar colisión con pared.

**T2.** Crear hechizo.

**T3.** Disparo simple.

**T3.1.** Mandar command a servidor.

**T3.2.** Ejecutar RPC para crear hechizo en el resto de clientes.

**T4.** Disparo combinado.

**T4.1.** Mandar command a servidor.

**T4.2.** Implementar algoritmo de ataque.

**T4.3.** Ejecutar RPC para crear hechizo combinado en el resto de clientes.

**T5.** Vida y maná.

**T5.1.** Implementar variables y funciones de control.

**T5.2.** Actualizar maná al lanzar hechizo.

**T5.3.** Actualizar salud al colisionar con hechizo de otro jugador.

**T5.4.** Implementar rutina de regeneración de maná.

**T6.** Implementar UI.

**T6.1.** Menú simple.

**T6.2.** Movimiento de cámara.

**T6.3.** UI de hechizo.

**T6.4.** UI de salud y maná.

**SPRINT 2.** Conseguir un combate simple.

En la figura 4.3 se presenta el planteamiento del sprint mediante un diagrama de Gantt y a continuación se detallan sus subtareas:

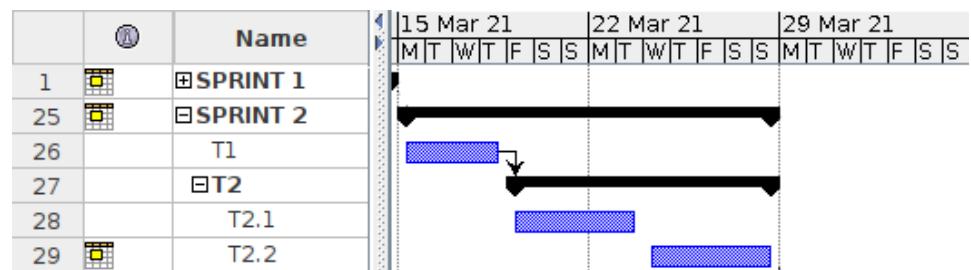


Figura 4.3: Diagrama de Gantt de la planificación del sprint 2

**T1.** Iniciar combate.

**T2.** Notificar al servidor de la derrota de un jugador.

**T2.1.** Mandar command a servidor.

**T2.2.** Registrar en el servidor que un jugador ha sido eliminado.

### SPRINT 3. Estructura de servidores (Servidor de combate y partida).

En la figura 4.4 se presenta el planteamiento del sprint mediante un diagrama de Gantt y a continuación se detallan sus subtareas:

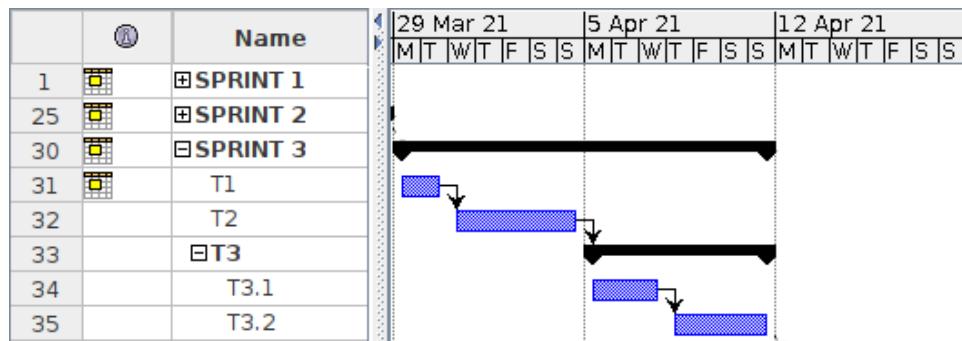


Figura 4.4: Diagrama de Gantt de la planificación del sprint 3

- T1.** Implementar variable de control de tipo de servidor.
- T2.** Funcionamiento base de servidor de partida.
- T3.** Creación de servidor de combate desde servidor de partida.
  - T3.1.** Script de creación y ejecución de script desde Unity.
  - T3.2.** Cambiar jugadores de conexión.

### SPRINT 4. Terminar combate.

Se presenta en la figura 4.5 el planteamiento temporal del sprint y a continuación se detallan las tareas que lo conforman:

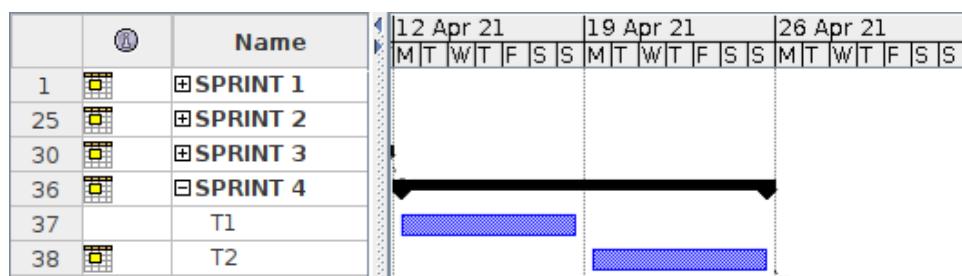


Figura 4.5: Diagrama de Gantt de la planificación del sprint 4

- T1.** Cambiar jugadores de conexión.
- T2.** Enviar mensaje de resultado a servidor de partida.

### SPRINT 5. Sistema de rondas y puntuación.

Se detalla el planteamiento temporal del sprint en la figura 4.6 y a continuación también las tareas que forman parte de él:

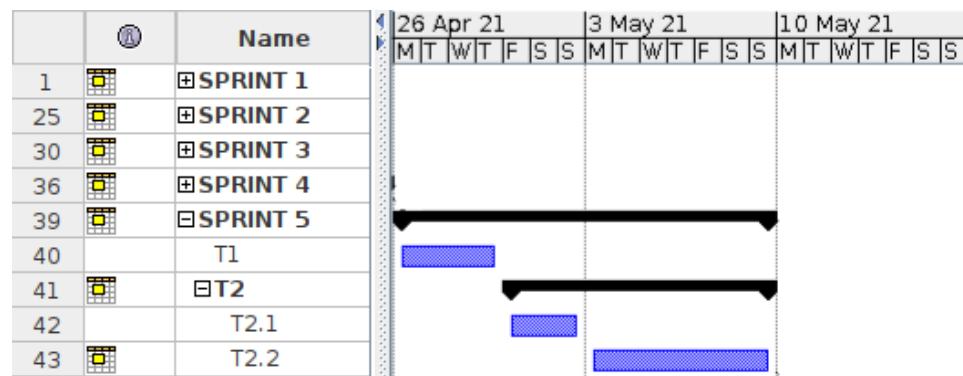


Figura 4.6: Diagrama de Gantt de la planificación del sprint 5

**T1.** Gestionar rondas al comenzar un combate desde el servidor de partida.

**T2.** ClasificationManager.

**T2.1.** Implementación base.

**T2.2.** Actualización al recibir mensaje de resultado de combate.

#### SPRINT 6. Servidor Matchmaking y comunicación

En la figura 4.7 se presenta el planteamiento del sprint mediante un diagrama de Gantt y posteriormente se detallan sus subtareas:

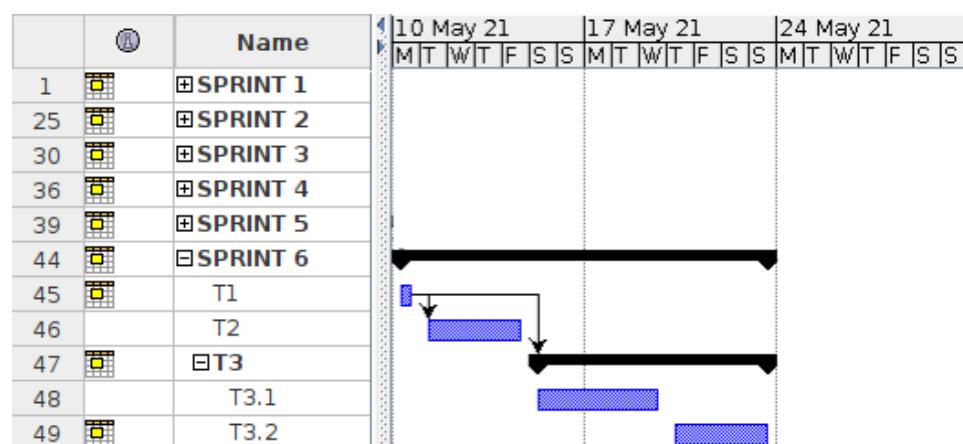


Figura 4.7: Diagrama de Gantt de la planificación del sprint 6

**T1.** Crear Servidor de Matchmaking.

**T2.** Crear Servidor de Partida desde Servidor de Matchmaking.

**T3.** Funcionamiento base.

**T3.1.** Implementación Matchmaker.

**T3.2.** Cambiar jugadores de conexión a servidor de partida.

#### SPRINT 7. MongoDB (Login/Registrar).

El planteamiento del sprint mediante un diagrama de Gantt está detallado en la figura 4.8. Posteriormente se detallan sus subtareas:

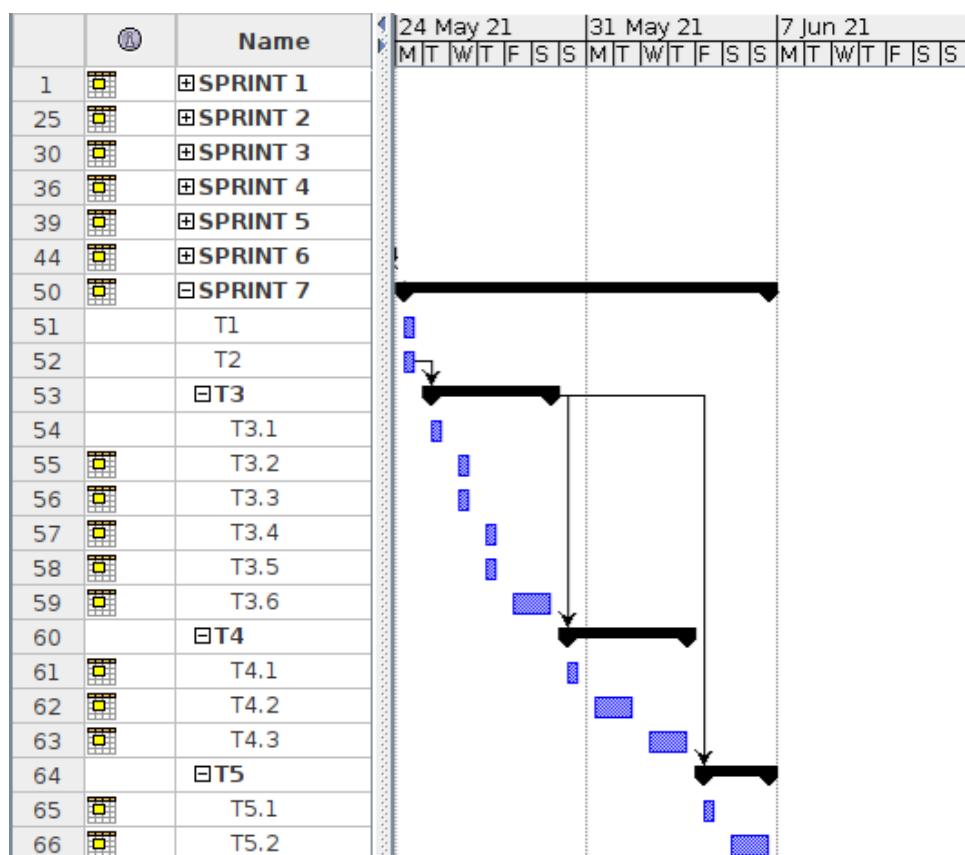


Figura 4.8: Diagrama de Gantt de la planificación del sprint 7

**T1.** Creación de cuenta en MongoDB.

**T2.** Creación de DB y colección de jugadores.

**T3.** Implementación de operaciones de BD con NodeJS.

**T3.1.** Instalación NodeJS.

**T3.2.** Implementación GET.

**T3.3.** Implementación POST.

**T3.4.** Implementación PUT.

**T3.5.** Implementación GET JWToken.

**T3.6.** Comprobación de funcionamiento usando 'curl'.

**T4.** Login.

**T4.1.** UI.

**T4.2.** Funcionamiento base.

**T4.3.** Seguridad adicional usando nonce.

**T5.** Registrar.

**T5.1.** UI.

**T5.2.** Funcionamiento.

#### SPRINT 8. Despliegue en Amazon Web Services (AWS).

En la figura 4.9 se presenta el planteamiento del sprint mediante un diagrama de Gantt y a continuación se detallan sus subtareas:

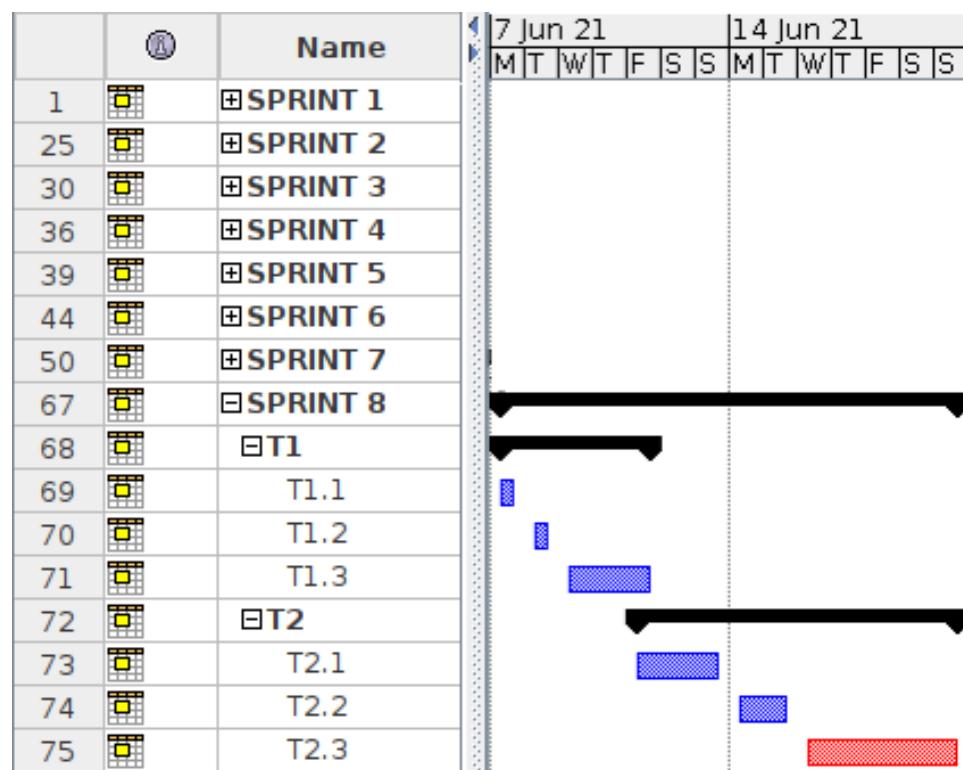


Figura 4.9: Diagrama de Gantt de la planificación del sprint 8

**T1.** Creación de máquina virtual en AWS:

**T1.1.** Creación de cuenta en AWS.

**T1.2.** Configuración de máquina virtual.

**T1.3.** Configuración del tráfico de router en máquina virtual.

**T2.** Subida del proyecto a la máquina virtual.

**T2.1.** Cambios de variables IP en el proyecto para ajustarse a la máquina virtual creada.

**T2.2.** Subida del proyecto usando FTP.

**T2.3.** Comprobación del funcionamiento usando SSH.

### 4.3. Estimación de costes

En el cuadro 4.1 se detalla el presupuesto en hardware del proyecto. En este caso, lo único necesario es un ordenador, y un router que de acceso a internet:

Presupuesto (hardware)			
Elemento	Precio (euros)	Cantidad	Total
Portátil Lenovo Legion Y520	799,99	1	799,99
Router	40,00	1	40,00
<b>Total</b>	—	—	839,99

Cuadro 4.1: Tabla de presupuesto de hardware del proyecto

En el cuadro 4.2 se detallan licencias de programas y demás recursos necesarios para el proyecto. Se ha buscado el mínimo coste posible, siendo de pago el paquete de sprites usado para los jugadores y el Wifi, esencial para el proyecto. Más específicamente, los paquetes de sprites usados provienen de usuarios en la plataforma *itch.io* [22] [23].

En el cuadro 4.3 se detalla una aproximación de lo que se cobraría por realizar este proyecto. La duración aproximada es de 4 meses (desde Marzo hasta Junio, ambos incluidos), lo que da 120 días, donde trabajando unas 6 horas de media al día nos da, finalmente, 720 horas de trabajo. Haciendo uso de páginas web en internet que recogen datos del sueldo medio de un programador junior [24], se observa un sueldo de 9,36 euros por hora.

Por tanto, se obtiene un salario en bruto de 6.739,2; es decir, unos 1.684,8 euros al mes.

Finalmente, en el cuadro 4.4 figura la suma del total obtenido en los anteriores cuadros.

Presupuesto (software)			
Elemento	Precio (euros)	Cantidad	Total por elemento
Ubuntu 20.04	0,00	1	0,00
Licencia Personal <i>Unity</i>	0,00	1	0,00
Licencia <i>framework Mirror</i>	0,00	1	0,00
Paquete recursos: <i>2D Pixel Dungeon</i>	0,00	1	0,00
Paquete sprites: <i>Animated Mages Character Pack</i>	1,34	1	1,34
Prueba 30 días <i>Visual Paradigm Visual Studio Code ProjectLibre</i>	0,00	1	0,00
Licencia <i>FREE Trello</i>	0,00	1	0,00
Despliegue en AWS	0,00	1	0,00
WiFi	35,00 al mes	4 meses	140,00
<b>Total</b>	—	—	141,34

Cuadro 4.2: Presupuesto software del proyecto

Salario	
Salario/Hora	9,36 [24]
Horas	720
<b>Total</b>	6.739,2

Cuadro 4.3: Presupuesto de salario

Presupuesto total	
Elemento	Precio
Hardware	839,99
Software	141,34
Salario	6.739,2
<b>Total</b>	7.720,53

Cuadro 4.4: Presupuesto total resumido del proyecto

#### 4.4. Desviación de las estimación

La estimación inicial no se cumplió al cien por ciento. Esto fue debido a varias razones:

- Ciertos aspectos del diseño inicial fueron cambiados en un momento posterior debido a nuevos conocimientos adquiridos, y otros aspectos necesarios para el proyecto no se tuvieron en cuenta en la planificación original por falta de conocimiento.

- Se tuvo que retrasar el desarrollo del proyecto cuando llegó la época de exámenes.

Estos sprints no fueron planificados originalmente, pero surgieron a partir de la primera razón descrita:

**SPRINT 8.** Rediseño del combate para favorecer la seguridad.

**T1.** Implementar movimiento con NetworkTransform.

**T2.** Comprobar funcionamiento correcto de colisiones.

**T2.1.** Separar collider de hechizo y collider de pared.

**T2.2.** Reasignar script de colisión con hechizos.

**T2.** Rediseñar disparo simple.

**T2.1.** Implementar el uso de NetworkServer.Spawn() para instanciar el hechizo.

**T3.** Rediseñar disparo combinado.

**T3.1.** Implementar el uso de NetworkServer.Spawn() para instanciar el hechizo.

**T4.** Reajustar el control de las variables de vida y maná.

**T1.1.** Cambiar estructura para dar control total al servidor.

**SPRINT 9.** Implementar uso de token para control de los jugadores.

**T1.** Mejorar la seguridad en la conexión.

**T1.1.** Comprobar de identidad de las conexiones.

**T2.** Implementar uso de token en combate.

**T2.1.** Usar token para identificar hechizos.

**T2.2.** Enviar mensaje de resultado de combate correctamente.

**T3.** Implementar uso de token en el servidor de partida.

**T3.1.** Implementar función para actualizar clasificación.

**SPRINT 10.** Arreglar UI.



# **Capítulo 5**

## **Diseño**

En este capítulo se profundiza en cómo se ha concebido el juego, tanto la idea base como en los diferentes pilares en los que se basa su funcionamiento.

### **5.1. Diseño del juego**

#### **5.1.1. Descripción del juego**

El juego desarrollado es un juego de acción con perspectiva cenital, donde cada jugador controla a un mago con cinco hechizos. Todos participan en una liga de forma individual durante varias rondas en las que combatirán entre ellos de forma aleatoria y los máximos clasificados combaten en una final para decidir el ganador de la partida. El número de rondas está determinado a tres rondas más la final. Se puede observar un boceto de cómo es visualmente en la figura 5.1.

El emparejamiento de los participantes en los combates se decide de forma aleatoria, y cada jugador obtiene puntos según en qué puesto queda en cada combate. Cada combate tiene una duración máxima de dos minutos.

Aparte de sus hechizos básicos, los magos pueden combinar dichos hechizos en grupos de hasta tres para conseguir ataques más potentes.

### **5.2. Descripción global del diseño de juego**

La arquitectura del juego en red basa su diseño en cuatro componentes: Servidor de matchmaking (emparejamiento), servidor de partida, servidor de combate, y cliente, los cuales se comunican entre ellos durante el transcurso de una partida, como se ve en la figura 5.1.

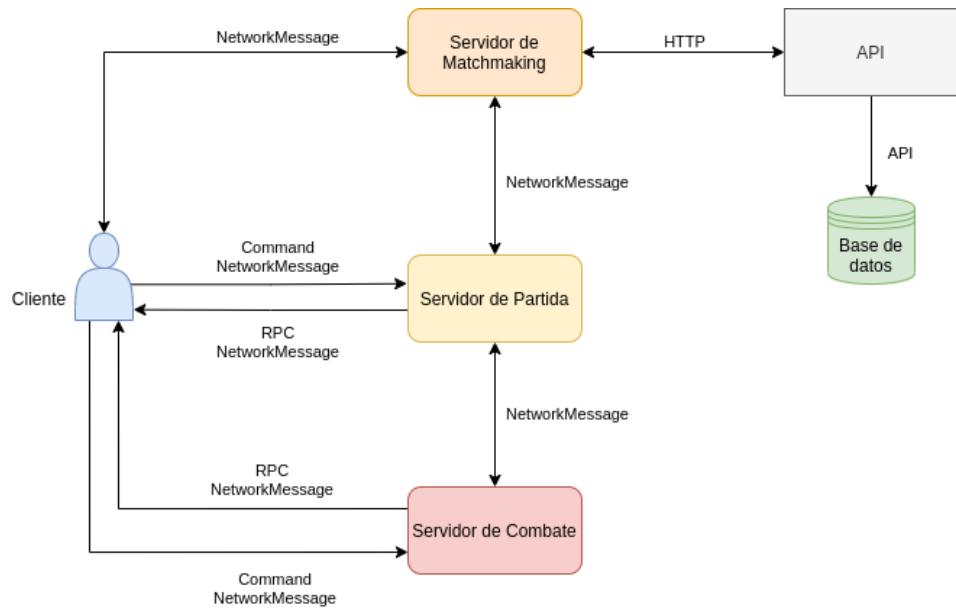


Figura 5.1: Diagrama de diseño

El funcionamiento de una partida será el siguiente: los jugadores ingresan en el sistema a través del servidor de matchmaking que, además de crear un token único para cada jugador, cuando sea oportuno, creará un servidor de partida y comunicará a los jugadores que han de cambiar su conexión a dicho servidor recién creado.

Cuando los jugadores se conecten al servidor de partida a través del cliente, obtendrán información sobre la partida que están jugando, y tendrán que comunicarle a este que están preparados para el siguiente combate. La primera vez que esto ocurre, el servidor proporcionará a los jugadores con un color específico para cada uno. Cuando todos los jugadores estén listos, el servidor de partida creará dos servidores de combate y comunicará a cada jugador dónde tiene que conectarse.

Estando conectados al servidor de combate, los jugadores deberán señalar que están preparados, y cuando todos en ese servidor lo estén, comenzará el combate. Tras este, el servidor de combate comunicará a los jugadores para que se cambien de nuevo al servidor de partida, que además recibirá los resultados de los combates de parte de los servidores que creó anteriormente.

Tras una cantidad determinada de rondas, se jugará una final entre los jugadores con mejor rendimiento, y tras esta final, la partida se considerará terminada. Cuando todos los jugadores abandonen el servidor de partida, este transmitirá el resultado de la partida al servidor de matchmaking que lo creó, el cual actualizará la información de cada jugador en la base de datos.

En las siguientes subsecciones se proporciona más información si es necesario en el diseño de cada uno de los componentes.

### 5.2.1. Servidor de matchmaking

Como se ha explicado anteriormente, es el encargado de gestionar las cuentas de los usuarios (Login/Registrar), de comenzar partidas y de actualizar los datos de los jugadores al terminar una partida.

#### Diagrama de clase

Se detalla en la figura 5.2 el diagrama de clase referente al diseño del servidor de matchmaking.

*Nota:* En Unity, el motor a usar en el proyecto, todas las componentes de un objeto heredan de la clase MonoBehaviour, con lo cual se ha obviado en la representación de diagrama de clases en pos de que el diseño sea generalizable. A su vez, todas las componentes de red heredan de NetworkBehaviour, requisito del framework de red elegido. Se han obviado ciertas funciones con el fin de que los diagramas sean legibles y aporten información útil sobre el diseño del proyecto.

### 5.2.2. Servidor de partida

Cuando es creado por el servidor de matchmaking, recibe como argumentos el puerto donde tiene que escuchar, el puerto límite que tiene disponible y los token de los jugadores que se deben conectar. De esta forma, si algún jugador ajeno intenta conectarse, el servidor podrá comprobar que no pertenece a la partida y rechazará dicha conexión.

Una vez todos los jugadores están conectados, el servidor les envía a todos la información referente a la tabla de clasificación para que cada cliente la muestre en pantalla.

Cuando todos los jugadores comunican que están preparados, el servidor de partida crea dos servidores de combate y comunica a los jugadores a qué servidor han de conectarse.

Después de cada combate, espera los resultados del servidor que creó, permitiéndole a este finalizar su ejecución y de nuevo enviando la nueva clasificación a todos los jugadores.

Cuando llega la hora de la final, tendrá constancia de qué jugadores deben jugar en ella y qué jugadores no, por lo que a estos últimos les envía un mensaje de que su partida se ha acabado y en qué puesto han quedado. Los otros jugadores continúan como hasta ahora.

En caso de que exista un empate en la clasificación que sea relevante para la final, se jugará otra ronda completa para intentar deshacerlo.

Una vez todos los jugadores que resten han abandonado el servidor tras la final, el servidor de partida se conecta al de matchmaking para enviarle el resultado de la partida que ha albergado, tras lo cual, con la aprobación del servidor de matchmaking, termina su ejecución.

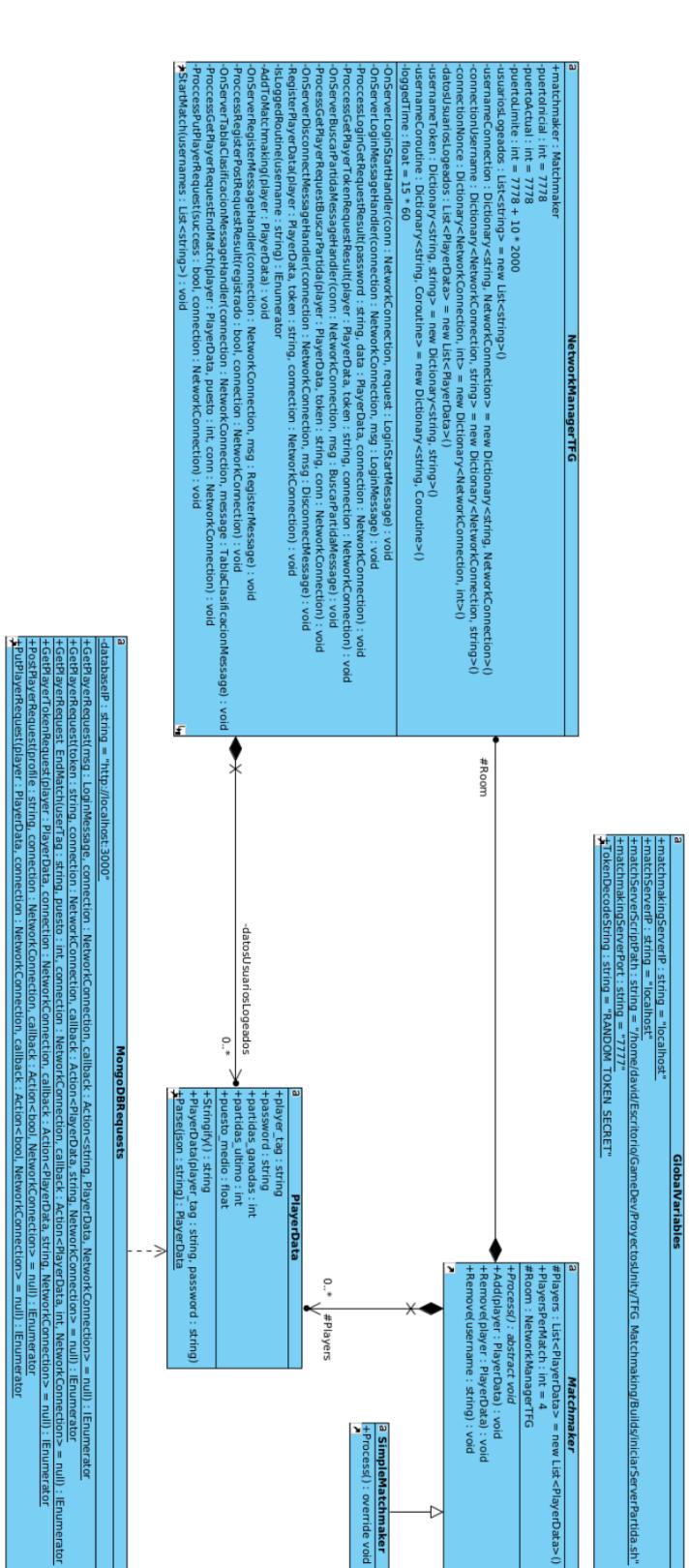


Figura 5.2: Diagrama de clase del servidor de matchmaking

### Diagrama de clase

Se detalla en la figura 5.3 el diagrama de clase correspondiente al diseño de esta instancia.

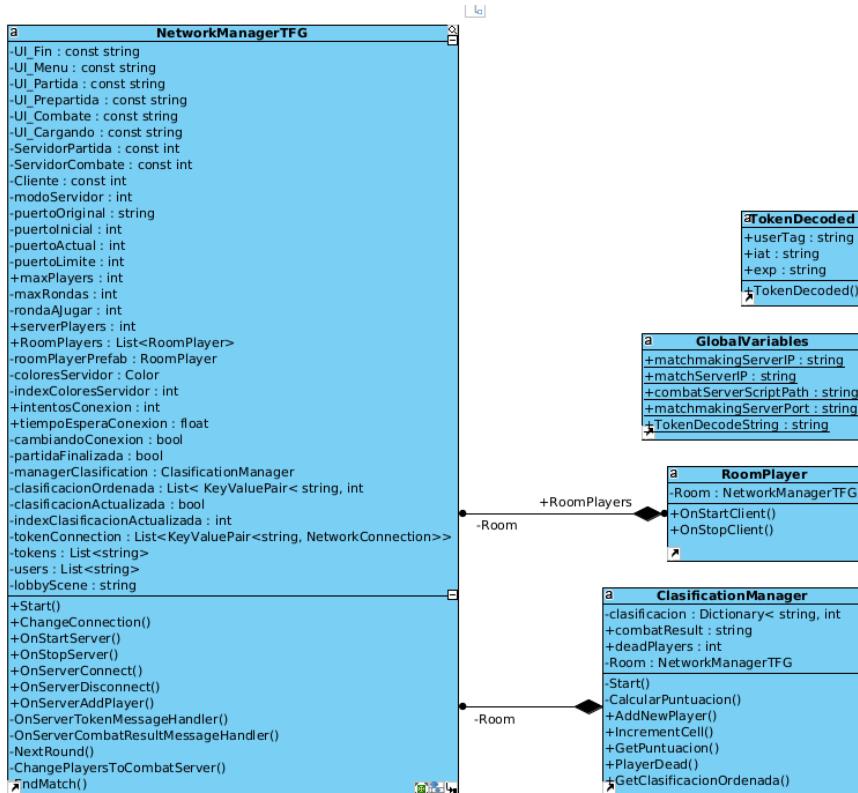


Figura 5.3: Diagrama de clase del servidor de partida

#### 5.2.3. Servidor de combate

Similar al servidor de partida, recibe como argumentos los token de los jugadores que van a conectarse a este servidor. Cuando los jugadores se conectan, el servidor les envía la información de los usuarios que participan en el combate para que los clientes la muestren en pantalla. Cuando los jugadores comunican que están preparados, el servidor empieza el combate.

Dentro del combate, el servidor se encarga de controlar el curso de este y las variables importantes como la salud y maná de los jugadores, el tiempo restante, etc. Esto protege al juego de posibles trampas que ocurrirían si el jugador controlase dichos aspectos.

Una vez terminado el combate, el servidor comunica a los jugadores que han de volver al servidor de partida, y manda a este el resultado del combate. Tras la aprobación del servidor de partida, finaliza su ejecución.

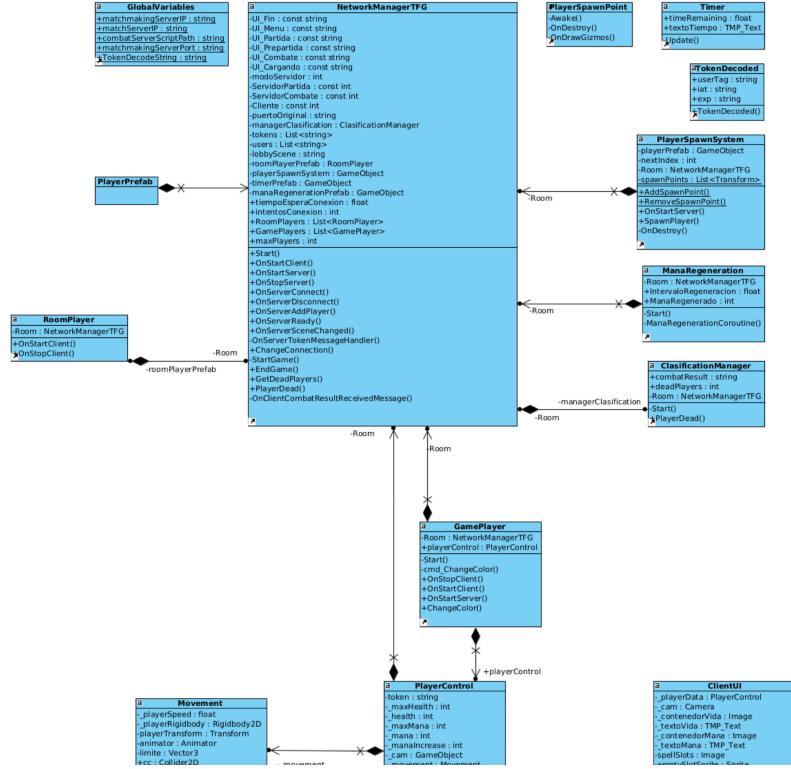


Figura 5.4: Diagrama de clase del servidor de combate (Primera mitad)

### Diagrama de clase

Se detalla el diagrama de clase del servidor de combate en la figura 5.4 y 5.5

#### 5.2.4. Cliente

### Diagrama de clases

Se observa en las figuras 5.6 y 5.7 el diagrama de clase referente al cliente.

## 5.3. Algoritmo de ataque

Los hechizos están diseñados de tal forma que se puedan agrupar de forma modular, incrementando enormemente la cantidad de hechizos que se pueden lanzar. Según en qué puesto se escogen, cada uno aportará de

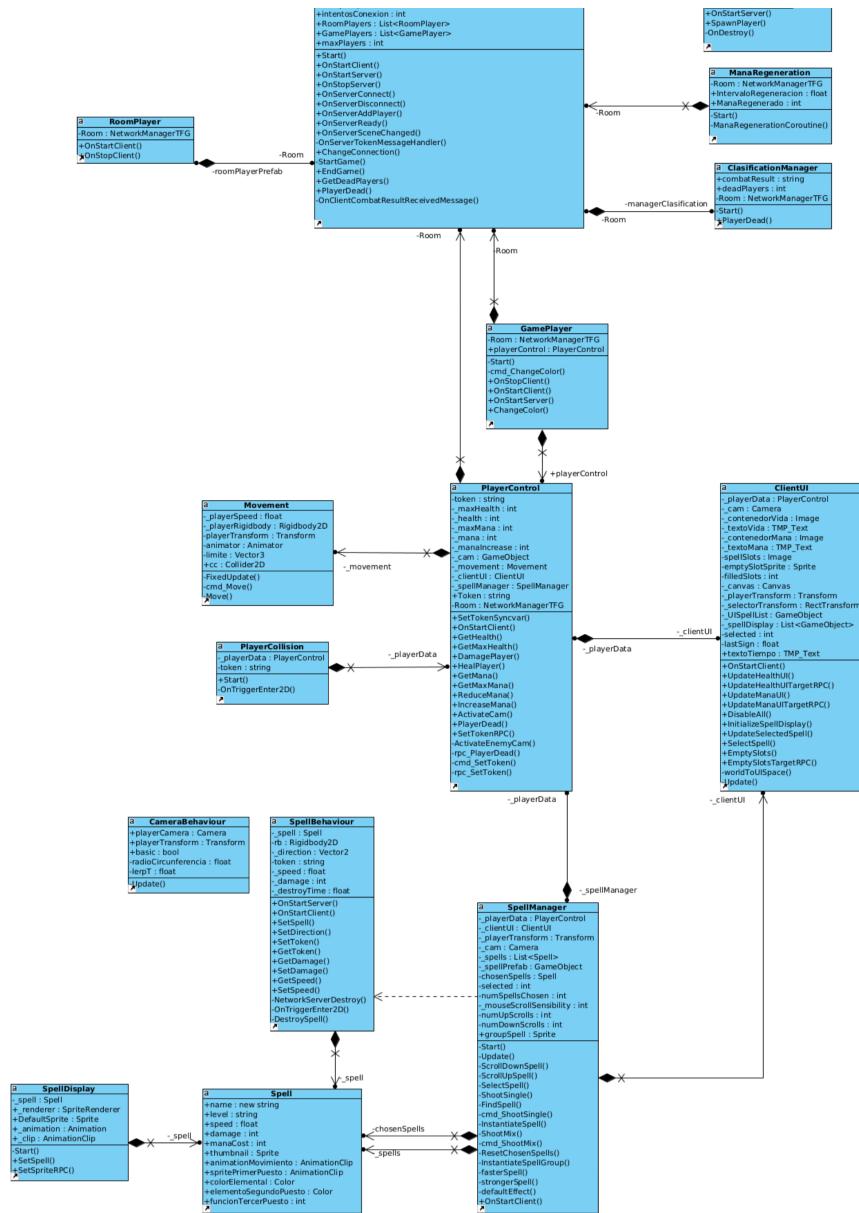


Figura 5.5: Diagrama de clase del servidor de combate (Segunda mitad)

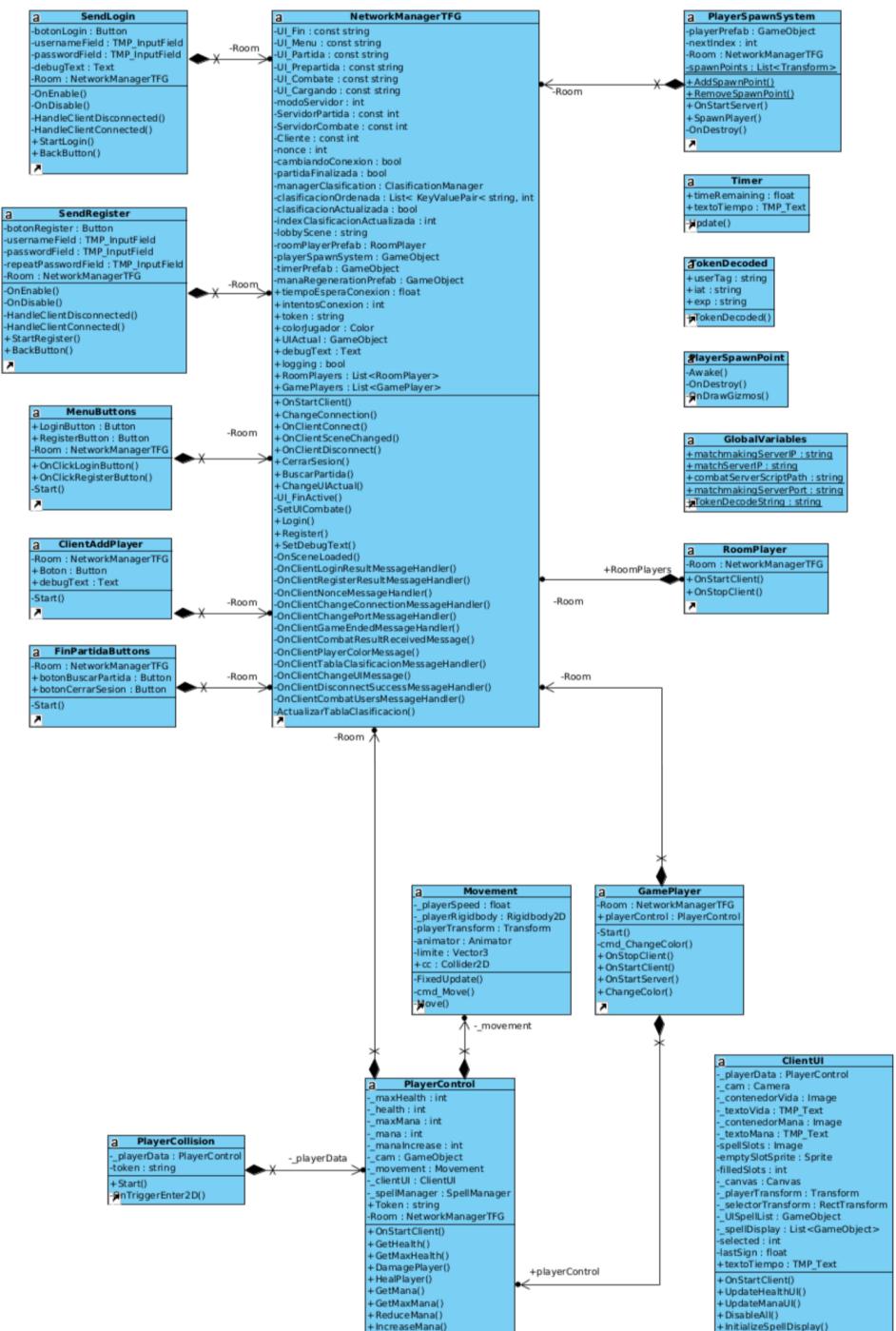


Figura 5.6: Diagrama de clase del cliente (Primera mitad)

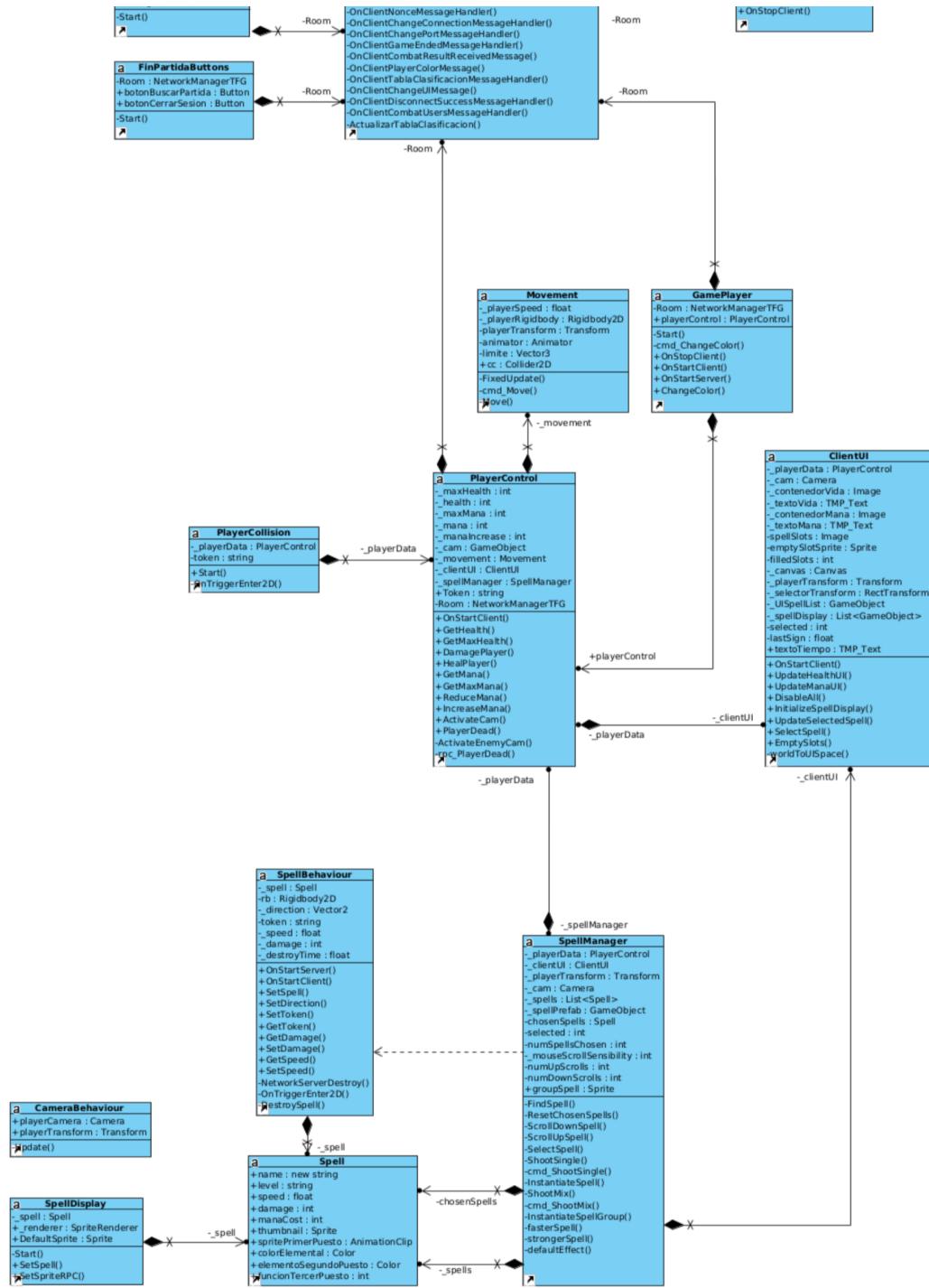


Figura 5.7: Diagrama de clase del cliente (Segunda mitad)

una forma u otra al poder del hechizo resultante. Así, la velocidad y daño nuevos se calculan según estas fórmulas, usando las características de los dos primeros hechizos:

$$\text{velocidad}_{\text{nueva}} = \frac{\text{velocidad}_{\text{hechizo1}} + \text{velocidad}_{\text{hechizo2}}}{2} \quad (5.1)$$

$$\text{daño}_{\text{nuevo}} = \text{daño}_{\text{hechizo1}} + \frac{\text{daño}_{\text{hechizo2}}}{2} \quad (5.2)$$

De esta forma, el hechizo resultante no tendrá una velocidad excesiva y el primer hechizo aportará más que el segundo ya que será el principal factor en daño.

En cuanto al tercer hechizo en el grupo, actúa de forma diferente y cada hechizo tendrá un efecto diferente: más velocidad, más daño, etc.

## 5.4. Comunicación entre las entidades

En esta sección se verá cómo las distintas entidades del juego se comunican entre sí, viendo los mensajes que se envían y cómo se han diseñado la comunicación en distintas funcionalidades del juego.

### 5.4.1. Actualización de posición de instancias

La actualización de posición será hecha desde el servidor hacia los clientes cada cierta cantidad de tiempo. Además, haciendo uso de componentes de Mirror, para que el movimiento sea lo más fluido posible usará interpolación por *snapshot*.

Un *snapshot* es una “captura” del estado de una simulación en un momento determinado. La interpolación por *snapshot* se basa en enviar *snapshot* cada cierto tiempo a los clientes y que se realice una interpolación lineal entre el estado en el que se encuentra dicho cliente y el estado recién recibido.

### 5.4.2. Mensajes

Aquí se explican los diferentes tipos de mensajes usados para comunicarse entre instancias.

#### **CombatResultMessage**

Usado para transmitir el resultado de un combate a un servidor de partida. Alberga los tokens de los jugadores en orden descendente de puesto.

**ResultReceivedMessage**

Confirmación de un servidor de que ha recibido un resultado que esperaba (ya sea de combate o de partida). Permite al servidor que recibe este mensaje finalizar su ejecución de forma segura.

**ChangeConnectionMessage**

Suministra información sobre la IP y el puerto donde se tiene que conectar el que lo recibe.

**LoginStartMessage**

Avisa al servidor de matchmaking de que un usuario quiere hacer login para que este le mande un NonceMessage.

**LoginMessage**

Proporciona al servidor de matchmaking con los datos de login (usuario y contraseña cifrada).

**RegisterMessage**

Mensaje que proporciona los datos de la nueva cuenta a crear al servidor de matchmaking.

**LoginResultMessage**

Comunica al usuario si el login ha sido satisfactorio. En caso afirmativo, este mensaje contendrá el token identificativo del usuario.

**RegisterResultMessage**

Comunica al usuario si el registro ha sido satisfactorio.

**NonceMessage**

Envía un nonce a un usuario. Un nonce es un número aleatorio de un solo uso, utilizado con el fin de proporcionar seguridad al login, puesto que permite que haya una comunicación exclusiva entre servidor y cliente. Se usa para la contraseña, ya que se envía un cifrado del cifrado de esta concatenado con el nonce.

**TokenMessage**

Envía un token identificativo de un usuario. Esto lo puede identificar en un servidor como usuario.

**GameEndedMessage**

Comunica que se ha terminado la partida a un usuario. Este mensaje puede enviarse antes de la final si un jugador no entra. Almacena el puesto en el que ha quedado ese jugador.

**PlayerColorMessage**

Almacena un color que va a usar un usuario y es específico a él. Así puede diferenciarse visualmente de otros jugadores.

**TablaClasificacionMessage**

Almacena la clasificación de la partida que se está jugando. Puede usarse para que los jugadores puedan ver esa información como para informar al servidor de matchmaking que se ha acabado la partida y el resultado de esta.

**ChangeUIMessage**

Avisa al usuario para que cambie la UI que está usando.

**CombatUsersMessage**

Da información sobre los usuarios presentes en un combate.

**BuscarPartidaMessage**

Comunica al servidor de matchmaking que un usuario está intentando buscar partida.

**DisconnectMessage**

Comunica al servidor de matchmaking que un usuario está intentando cerrar sesión.

**DisconnectSuccessMessage**

Asegura al usuario de que puede cerrar sesión de forma segura.

### 5.4.3. Funcionalidades

Haciendo uso de los mensajes descritos anteriormente, así se llevan a cabo las siguientes funcionalidades:

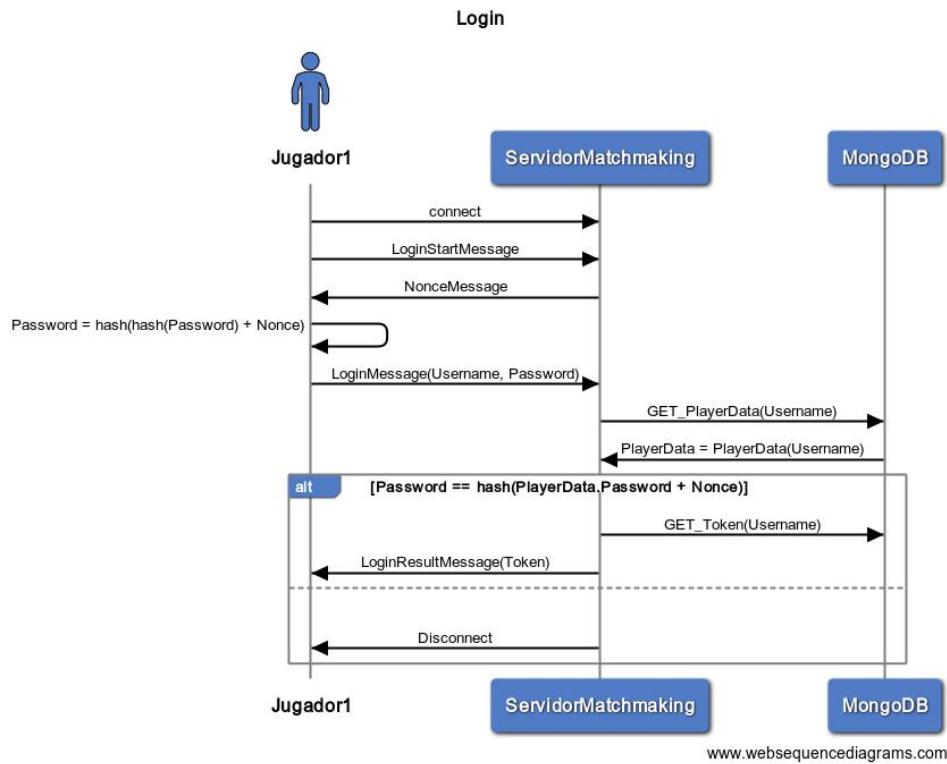


Figura 5.8: Diagrama de secuencia del proceso de Login

## Login

Como se describe en la figura 5.20, el jugador envía un mensaje del tipo LoginStartMessage al servidor de matchmaking para comenzar el proceso, este le envía un nonce (definido en el apartado 5.4.2) de vuelta con un mensaje del tipo NonceMessage. El jugador concatena dicho nonce con su contraseña cifrada, y cifra de nuevo dicho resultado. A través de un LoginMessage, envía al servidor los datos de login, que comprueba que son correctos haciendo una consulta a MongoDB. Finalmente, si el resultado es correcto, el servidor de matchmaking obtendrá y enviará al usuario su token personal con un LoginResultMessage, y en caso contrario, cortará la conexión.

El procedimiento para hacer login usando un nonce se realiza para tener un login lo más seguro posible, como se pedía en la sección de requisitos no funcionales del capítulo 3.

## Registrar

Como se detalla en la figura 5.21, el usuario cifra la que quiere que sea su contraseña y la envía junto a su nombre de usuario deseado en un

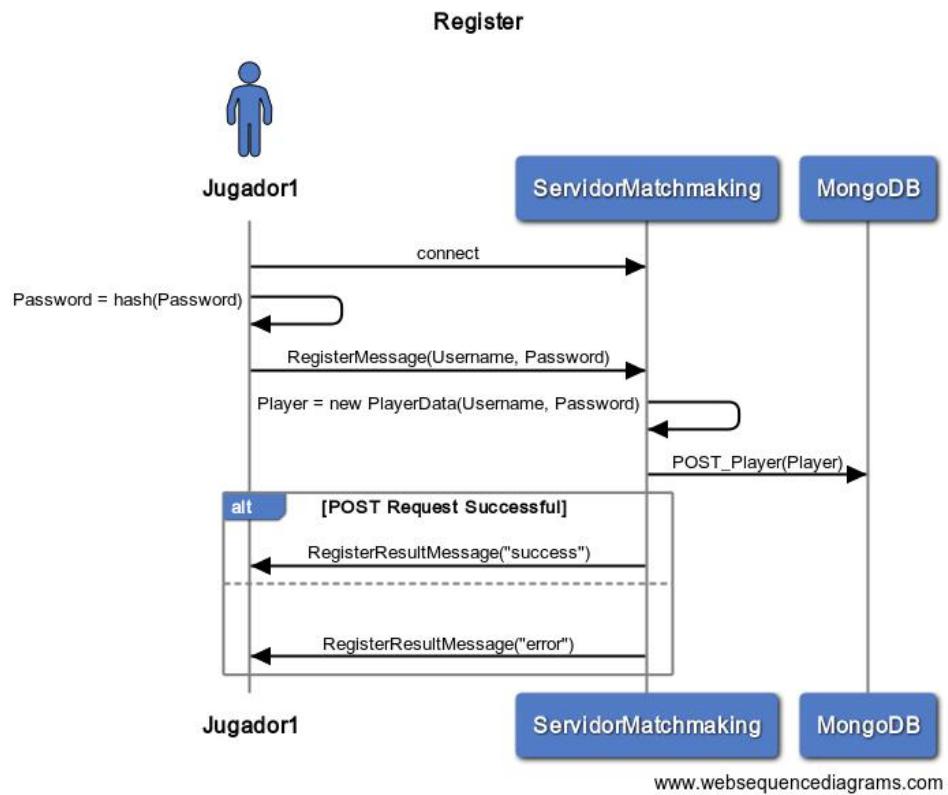


Figura 5.9: Diagrama de secuencia del proceso de registro

RegisterMessage. El servidor de matchmaking realizará una petición POST a MongoDB y comunicará el resultado al jugador.

#### Validar presencia jugador (Servidor Partida)

Como se detalla en la figura 5.10, el jugador manda un TokenMessage con su token al servidor de partida, el cual comprobará que dicho token se encuentra en la lista de los posibles clientes. En caso contrario desconectará al jugador. En caso de ser correcto, el servidor de partida enviará un ChangeUIMessage para que el usuario muestre una pantalla de carga previa a otro mensaje con el siguiente paso del jugador. Tras procesar el estado en el que se encuentra el juego y el jugador dentro de este, mandará un mensaje del tipo TablaClasificaciónMessage, en caso de que siga jugando, o uno del tipo GameEndedMessage si la partida ha finalizado para el jugador.

#### Validar presencia jugador (Servidor Combate)

Como se representa en la figura 5.11, el jugador, tras conectarse al servidor de combate, envía un mensaje del tipo TokenMessage con su token.

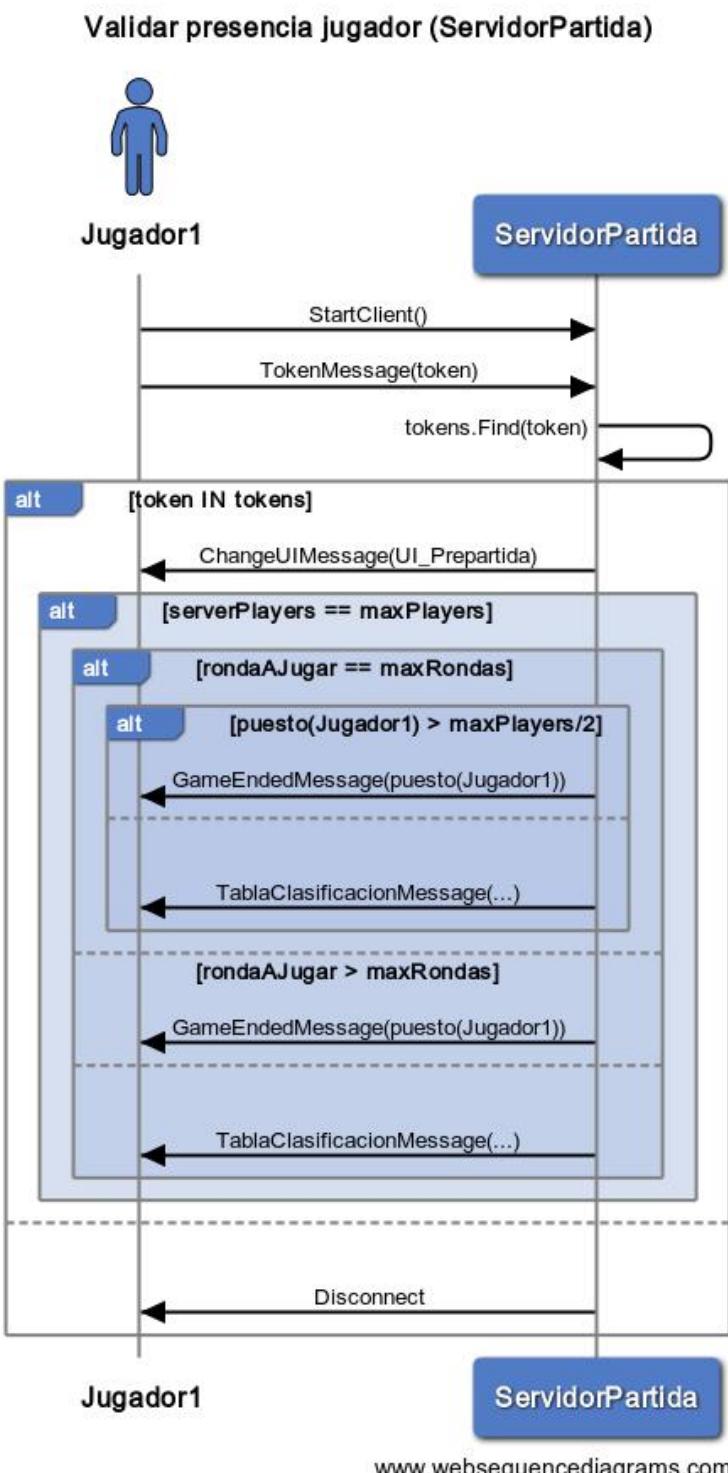


Figura 5.10: Diagrama de secuencia para validar la presencia de un jugador en servidor de partida

El servidor lo recibe y comprueba si entra en la lista de tokens admitidos para ese servidor. En caso afirmativo, comunicará al jugador la información del combate a disputar mediante un mensaje CombatUsersMessage. En caso contrario, desconectará al jugador.

### Cambiar Conexión

Como describe la figura 5.12, el servidor (del tipo que sea), envía un mensaje del tipo ChangeConnectionMessage, con los datos a usar. El jugador se desconecta del servidor, cambia los datos de la conexión de acuerdo a este mensaje, y comienza su conexión de nuevo.

### Informar de resultado de combate

Como se detalla en la figura 5.13, el servidor de combate, tras comunicar a todos los jugadores de que han de cambiar de conexión, se conecta al mismo servidor de partida para enviar un mensaje del tipo CombatResultMessage, donde almacena el resultado del combate. Espera a que el servidor de partida haya recibido el resultado, notificado por un mensaje ResultReceivedMessage, y termina su ejecución.

### Informar de resultado de partida

Como se detalla en la figura 5.13, el servidor de partida, una vez todos los jugadores han abandonado el servidor, se conecta al servidor de matchmaking para enviar un mensaje del tipo TablaClasificacionMessage, donde almacena el resultado de la partida. Espera a que el servidor de matchmaking haya recibido el resultado, notificado por un mensaje ResultReceivedMessage, y termina su ejecución.

### Asignar color a jugador

Como viene explicado en la figura 5.15, el jugador, tras haber validado su presencia en el servidor de partida y haber notificado que está preparado para jugar, recibe un mensaje PlayerColorMessage si no lo ha recibido antes. En este mensaje se recibe el color a usar para el personaje del jugador.

### Buscar Partida

Tal y como se explica en la figura 5.16, tras conectarse al servidor de matchmaking, el jugador envía un mensaje del tipo BuscarPartidaMessage con su token. El servidor entonces comprueba que es un usuario logeado mediante su token y, en caso afirmativo, obtiene los datos del jugador de MongoDB para registrarlos de nuevo. En caso de error o de que no sea un usuario logeado, desconecta esa conexión.

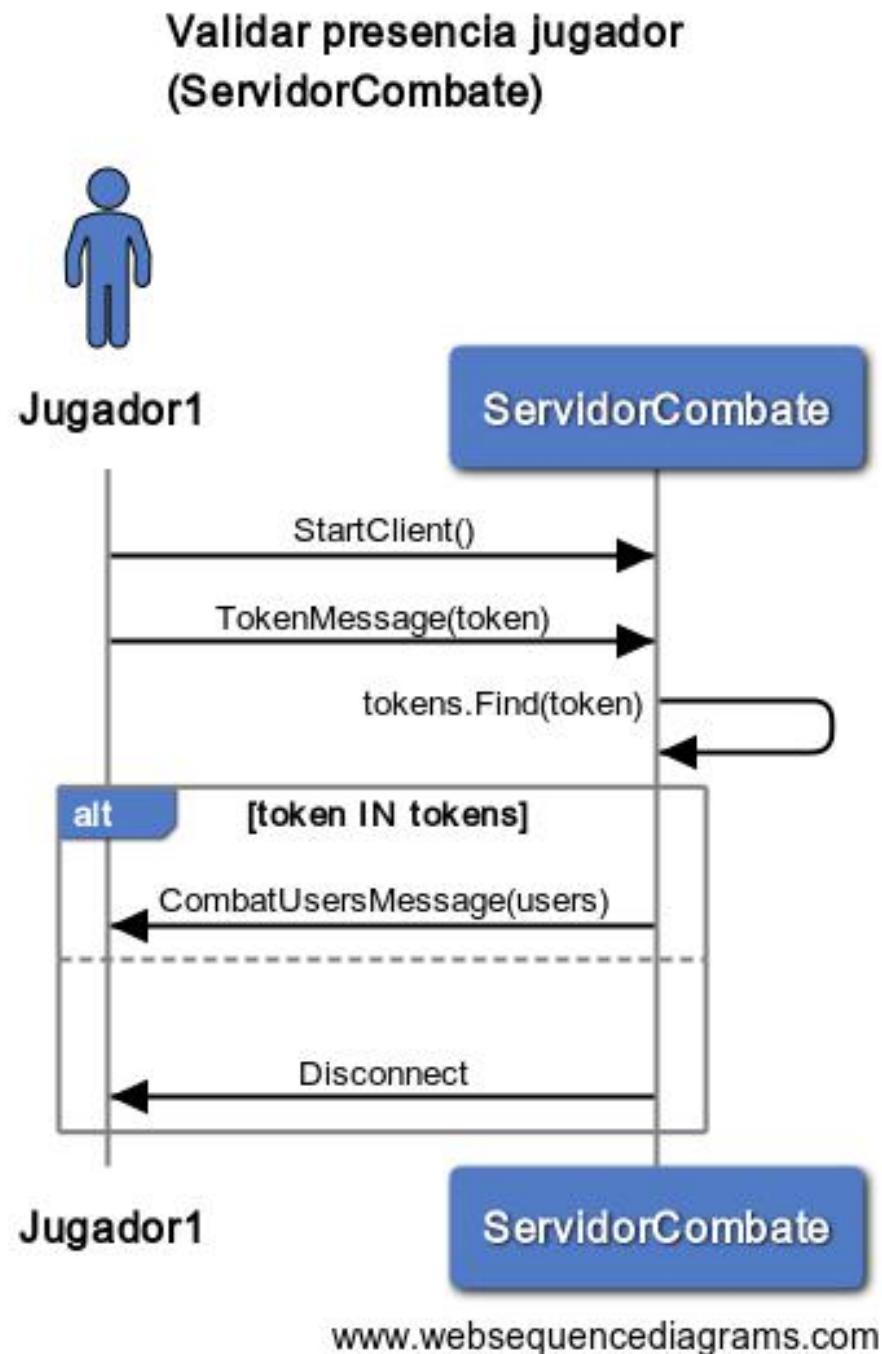


Figura 5.11: Diagrama de secuencia para validar la presencia de un jugador en servidor de combate

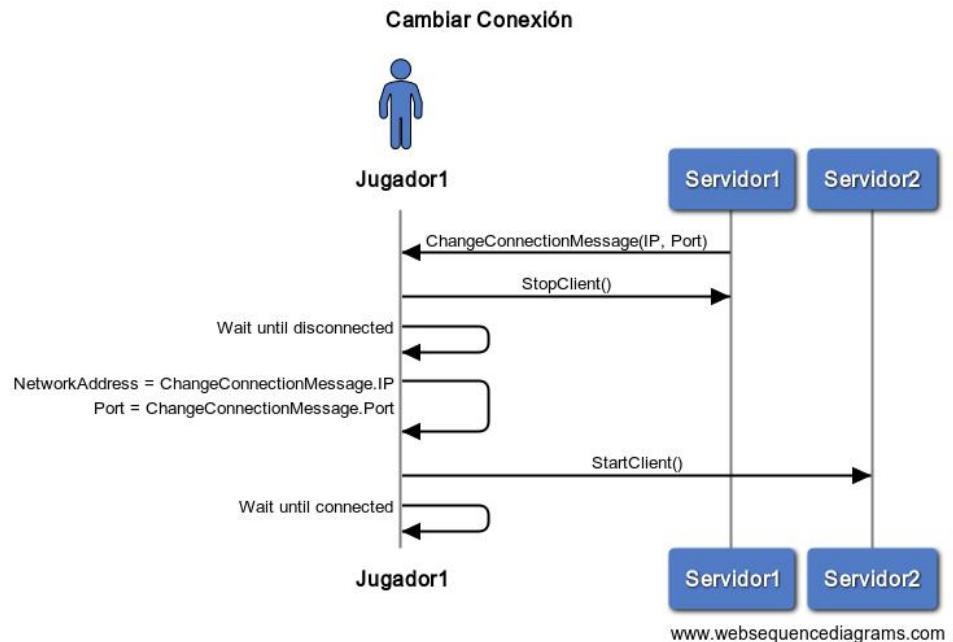


Figura 5.12: Diagrama de secuencia sobre cambiar conexión

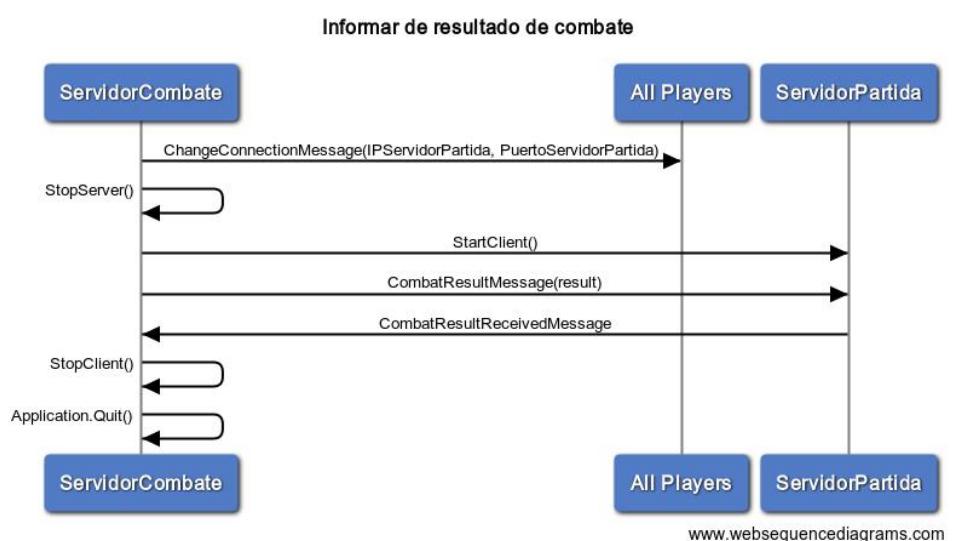


Figura 5.13: Diagrama de secuencia sobre informar de resultado de combate

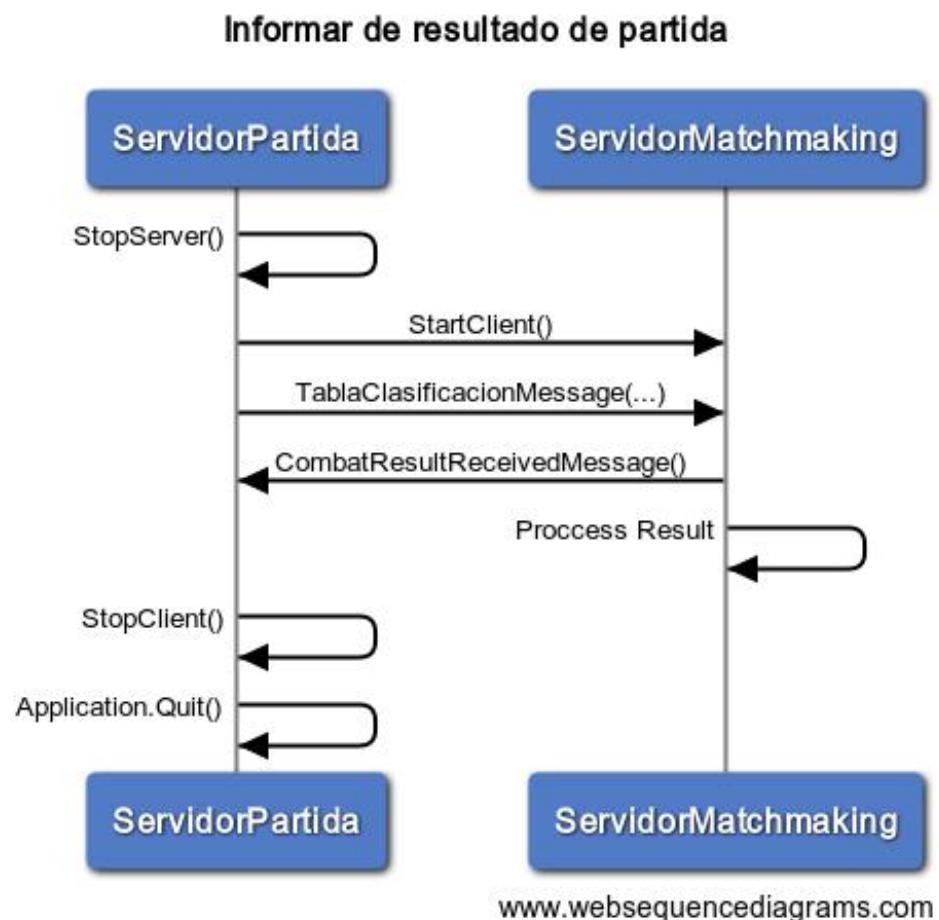


Figura 5.14: Diagrama de secuencia sobre informar de resultado de partida

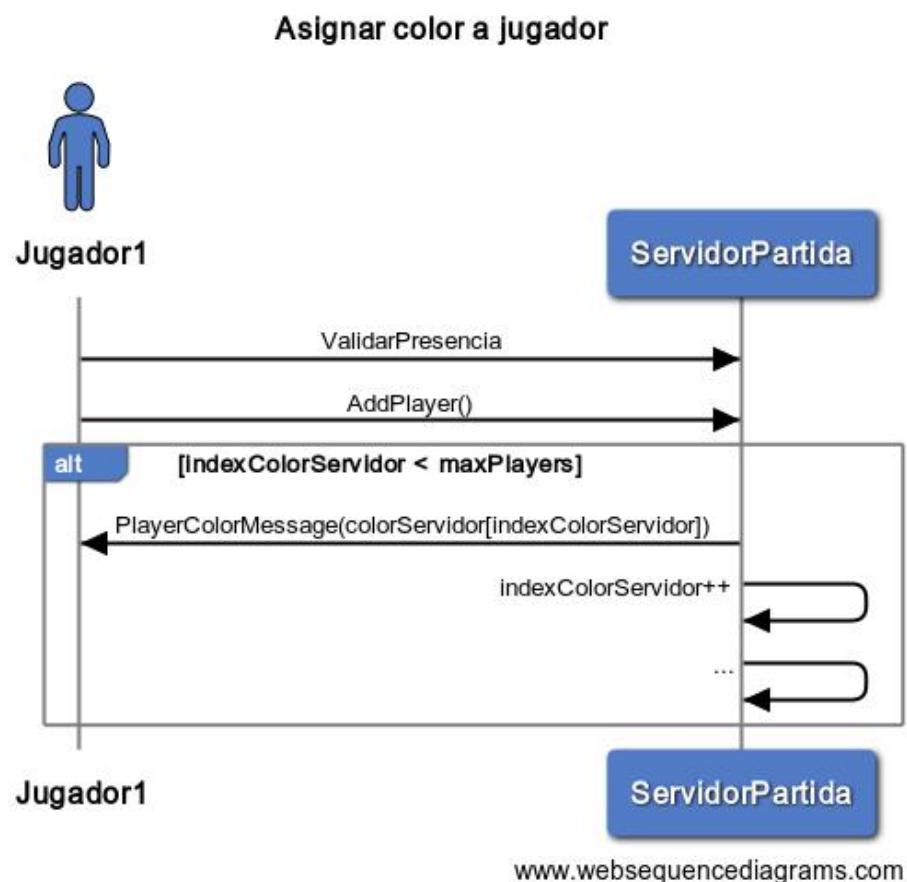


Figura 5.15: Diagrama de secuencia sobre asignar color a un jugador

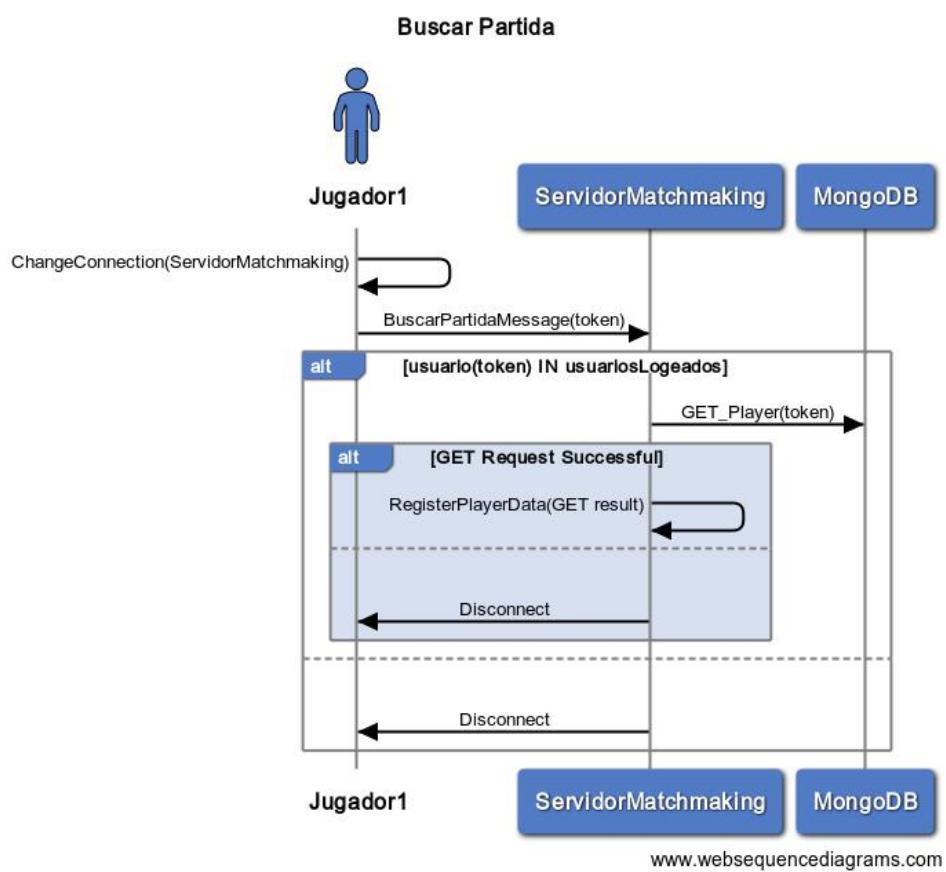


Figura 5.16: Diagrama de secuencia sobre buscar partida

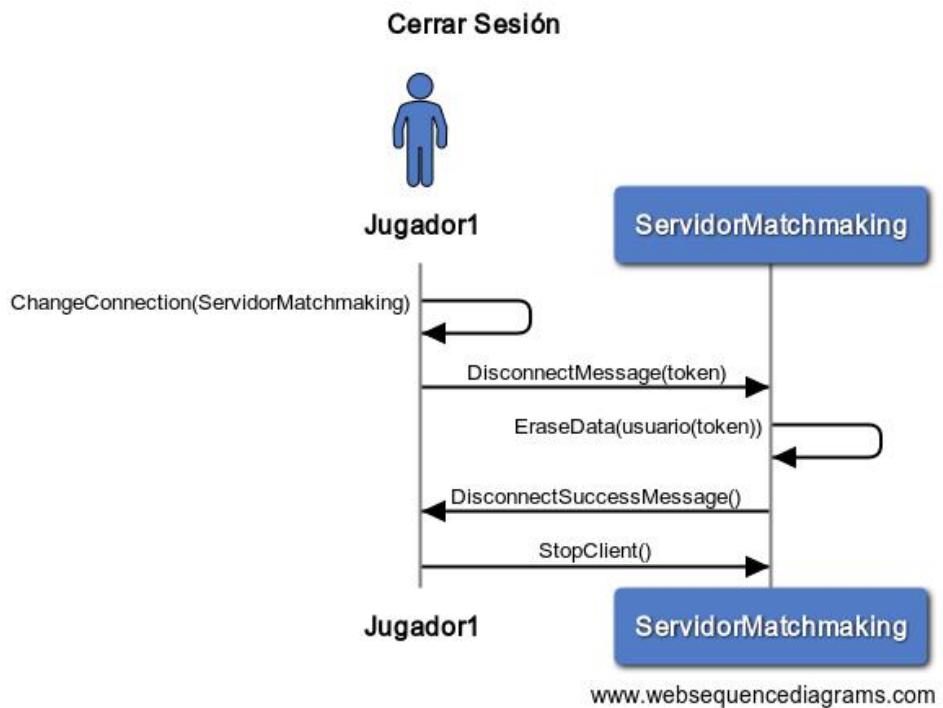


Figura 5.17: Diagrama de secuencia sobre cerrar sesión

### Cerrar Sesión

Como se detalla en la figura 5.17, el jugador, tras conectarse al servidor de matchmaking, envía un mensaje `DisconnectMessage` con su token. Esto provoca que el servidor borre todos los datos de su sesión que tenía almacenados y envíe de vuelta un mensaje `DisconnectSuccessMessage`. Esto sirve como confirmación de la operación para el jugador, que se desconecta del servidor.

## 5.5. Interfaz

Para el proyecto he buscado una interfaz simple, que dé la información necesaria sin excederse. En esta sección se verán los bocetos de las interfaces en cada escena del juego.

### 5.5.1. Diagrama de flujo entre pantallas

En la figura 5.18 se muestra las transiciones entre distintas interfaces para tener una representación más visual del proyecto.

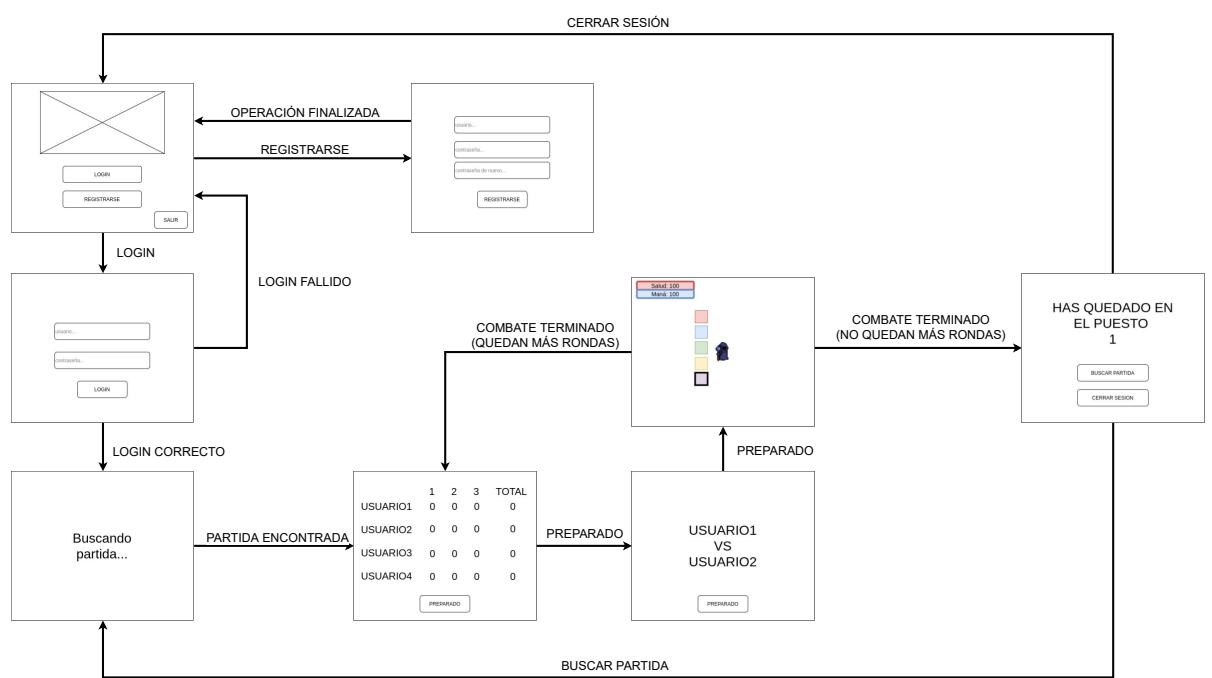


Figura 5.18: Diagrama de flujo de las interfaces

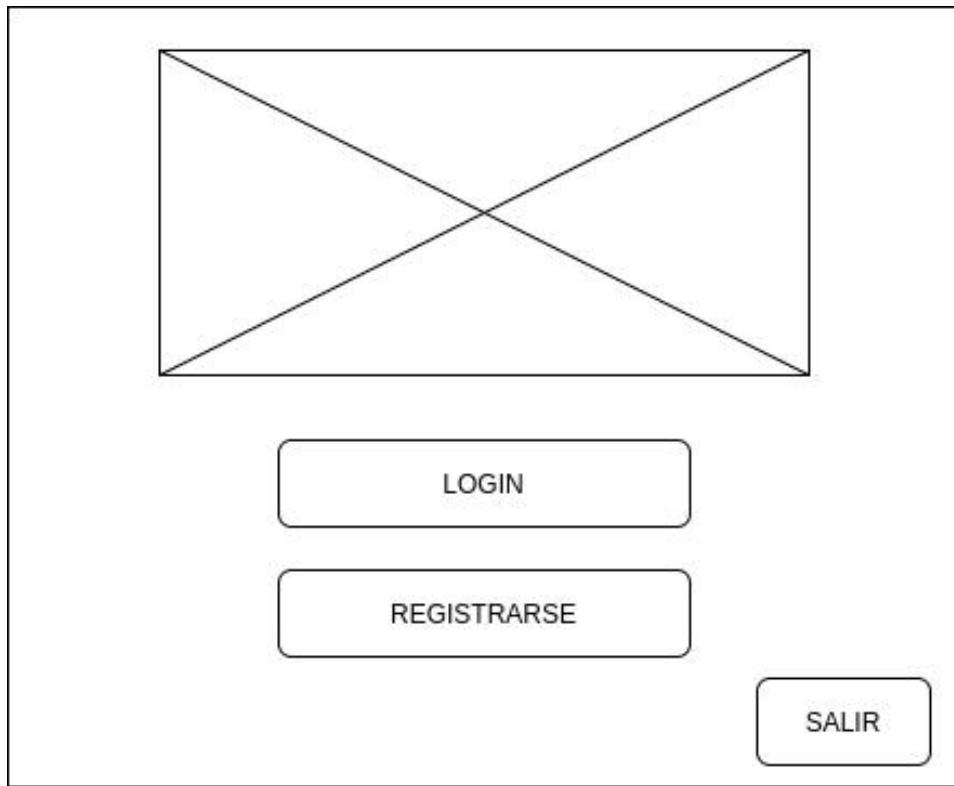


Figura 5.19: Boceto de interfaz de menú

### 5.5.2. Interfaz de menú

Se muestran los bocetos referentes al menú previo a entrar a una partida.

#### Menú principal

Para el menú principal, como se observa en la figura 5.19, se muestra el logo del juego, y el usuario tiene las opciones de loguearse, registrarse o salir del juego.

#### Interfaz de Login

Como se detalla en la figura 5.20, para el login, el usuario debe introducir su nombre y contraseña.

#### Interfaz de Registro

En el caso de registrarse, se detalla en la figura 5.21 que el usuario deberá introducir dos veces la contraseña a tener, tal y como figura en el apartado 3.2

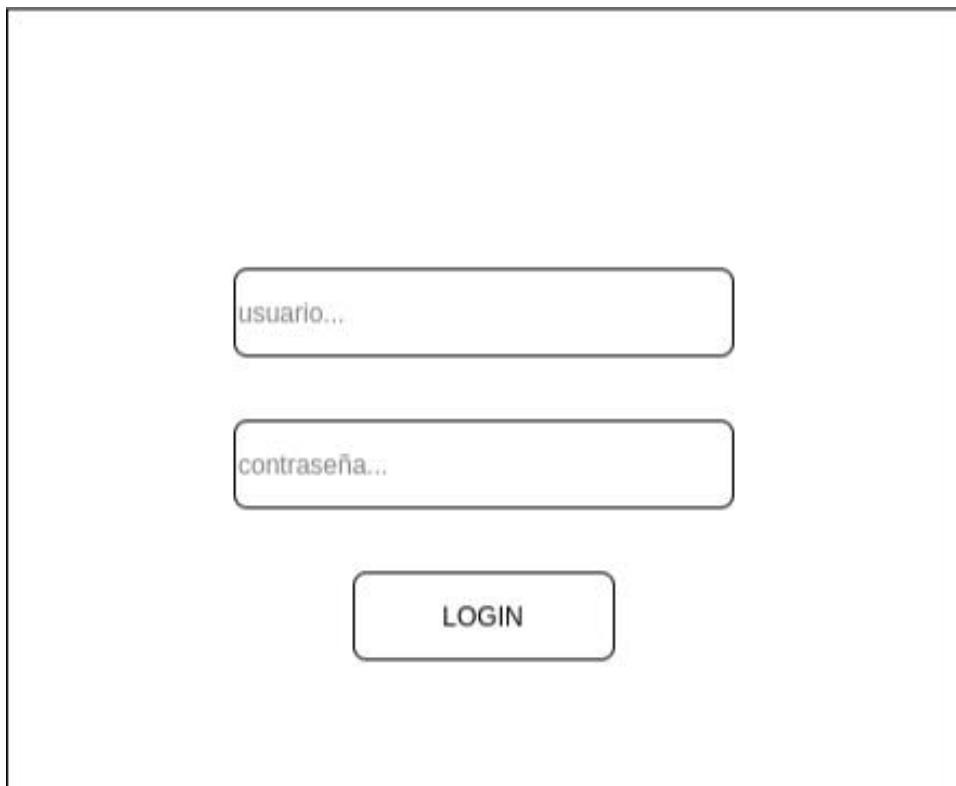


Figura 5.20: Boceto de interfaz de login



Figura 5.21: Boceto de interfaz de registro

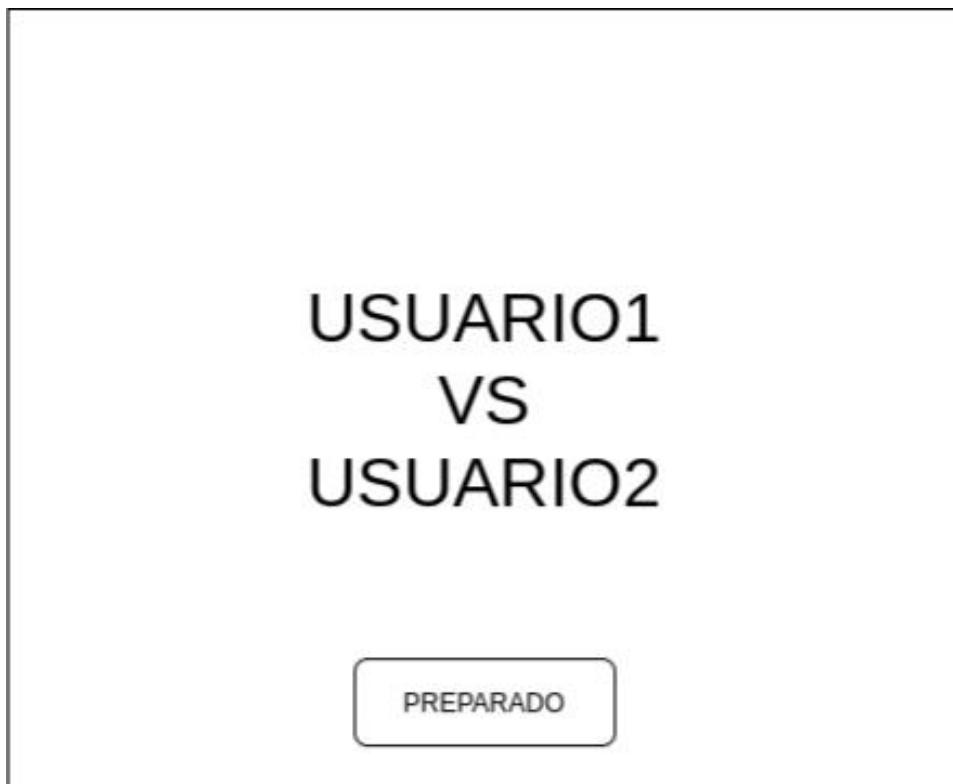


Figura 5.22: Boceto de interfaz de pre-combate

### 5.5.3. Interfaz de combate

Se muestran los bocetos referentes al combate.

#### Interfaz pre-combate

En esta pantalla, detallada en la figura 5.22, se muestra los usuarios a participar en el combate, y un botón para que el usuario comunique que está preparado.

#### Interfaz de combate

En el combate, como se muestra en la figura 5.23, se muestra tanto la salud como el maná del usuario, información importante para el usuario, y para los hechizos, se muestran al lado de su avatar. Esto se debe a que, al ser un juego de cierta rapidez en el movimiento y al actuar, apartar la mirada podría ser perjudicial para los jugadores ya que su atención iría hacia otro punto, por lo que de esta forma el jugador puede seguir mirando hacia su avatar y sus alrededores, sacrificando algo de visión. El hechizo que esté seleccionado actualmente tiene una caja negra alrededor.

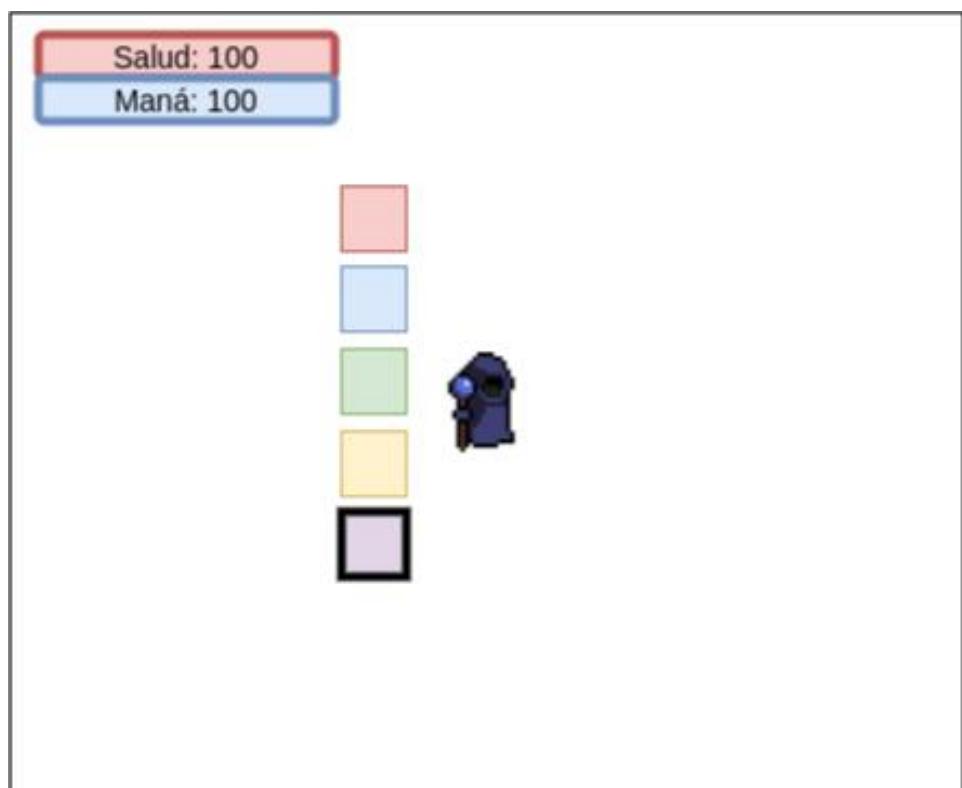


Figura 5.23: Boceto de interfaz de combate

	1	2	3	TOTAL
USUARIO1	0	0	0	0
USUARIO2	0	0	0	0
USUARIO3	0	0	0	0
USUARIO4	0	0	0	0

**PREPARADO**

Figura 5.24: Boceto de interfaz entre rondas

#### 5.5.4. Interfaz de partida

Se muestran los bocetos referentes a la partida fuera del combate.

##### Interfaz entre rondas

Para la pantalla entre rondas, la información más relevante al jugador es cómo se encuentra la partida en ese punto, es decir, la clasificación actual, como se observa en la figura 5.24.

##### Interfaz de partida finalizada

Por último, como se detalla en la figura 5.25, al finalizar la partida es necesario que el jugador sepa en qué posición ha acabado, y darle las opciones que necesita: buscar otra partida, o cerrar sesión para terminar el juego.



Figura 5.25: Boceto de interfaz de partida finalizada

# Capítulo 6

# Implementación

En este capítulo se profundiza en cómo se ha llevado a cabo el diseño mostrado en el capítulo 5.

## 6.1. Base de datos de jugadores

En este apartado se habla sobre cómo se creó la base de datos usada en el proyecto y cómo se consigue consistencia de datos siguiendo la estructura del proyecto.

### 6.1.1. Base de datos

Para la base de datos, se usa MongoDB [25], puesto que proporciona una forma rápida y gratuita de obtener una base de datos. Para implementar la API del backend, se utiliza Node.js [26], usando el framework Express.js, muy útil para crear APIs HTTP [27].

### 6.1.2. Comunicación con MongoDB

La comunicación con la base de datos se realiza a través de la clase de Unity UnityWebRequest, que permite realizar peticiones HTTP a la API y obtener datos de MongoDB.

### 6.1.3. Consistencia de datos

Una de las desventajas de la estructura de servidores detallada en el capítulo 5 es que, al tener a los jugadores cambiando de servidor, es necesario guardar algún tipo de referencia de ellos en el servidor de matchmaking, para saber qué usuarios están logueados y cuáles no lo están. En este caso, dicha referencia será su nombre de usuario, que se almacenará en el servidor durante veinte minutos, una cantidad de tiempo que excede al de una partida para dar una ventana de tiempo en caso de que el usuario tarde en volver

al servidor de matchmaking. Cuando el usuario vuelve al servidor, si busca una nueva partida, esta cuenta de 20 minutos se renueva.

## 6.2. Identificación de los jugadores

Para poder identificar a los jugadores en varios servidores, se hace uso de *Json Web Tokens* (JWT), obtenidos a partir de la API de MongoDB anteriormente explicada. Para poder manejar esto tokens en el proyecto de Unity, se utiliza la librería *jwt-dotnet* [28], que ha permitido poder acceder a los datos de los token.

## 6.3. Creación de servidores

Para la creación de servidores, se hace uso de la clase *System.Diagnostics.Process*, usada para ejecutar comandos. En este caso, se ejecuta un script para crear servidores de partida y otro para crear servidores de combate.

### 6.3.1. Script de creación

En dicho script se usa la herramienta *ssh* de OpenSSH, la cual permite ejecutar comandos de forma remota en otras máquinas. Esto permite la posibilidad de usar varias máquinas con distintos tipos de servidores, lo que mostraría el potencial en escalabilidad de la estructura propuesta en el capítulo anterior.

Al usar *ssh* con otra máquina, se pide introducir una contraseña antes de realizar cualquier acción, por lo que se tiene que automatizar este proceso para usarlo desde Unity. Para ello, se genera un par de claves (pública y privada) a modo de autenticación, usando la herramienta *ssh-keygen*. En la máquina donde se quiere hacer login, se concatena la clave pública a sus claves autorizadas. Finalmente, ya se podría usar *ssh* sin necesidad de contraseña.

Al servidor a crear se le mandan datos relevantes a su ejecución:

- Puerto en el que escuchar.
- IP y puerto del servidor que lo creó.
- Modo del servidor creado.
- Tokens de los jugadores a conectarse.

En el listado 6.1 se ve un ejemplo de los guiones usados para crear servidores.

Listing 6.1: Script de creación de servidores

```
#!/bin/bash
ip=$1
shift
ssh $ip export TERM=xterm && $RUTA_EJECUTABLE_SERVIDOR "$@"
```

## 6.4. Esperas dentro de un proceso

En ciertos procedimientos del proyecto, es necesario que servidor o cliente esperen antes de realizar alguna acción para asegurar el éxito de esta:

- Cuando todos los jugadores envían su token al servidor de partida, se necesita tener la clasificación actualizada antes de informar a los jugadores del siguiente paso a tomar. Para asegurar el correcto funcionamiento del servidor, este espera durante un tiempo determinado en intervalos cortos comprobando si la clasificación está ya actualizada o no.
- Cuando un jugador cambia de conexión, espera de forma similar al ejemplo anterior para asegurar que está desconectado y después conectarse a otro servidor, durante lo cual también esperará para completar el proceso antes de continuar.
- Al intentar hacer login, el cliente espera para asegurar que tiene el nonce necesario para completar el procedimiento.
- Al acabar un combate, el servidor espera tras enviar un mensaje de cambio de conexión para asegurar que este se manda correctamente.

Esta espera se realiza a través de *Coroutines* de Unity. Al contrario que las funciones normales, las cuales han de ejecutarse de principio a fin sin parar, las *Coroutines* de Unity son funciones con la capacidad de parar su ejecución y devolver el control a Unity y seguir en otro momento. El momento de la espera se especifica a través de la sentencia *yield return*, donde se puede concretar el tiempo a esperar.

## 6.5. Interfaz de usuario

Para representar interfaces en Unity, se usa un objeto con la componente *Canvas* (Al objeto se le llamará Canvas, y salvo que se especifique lo contrario, se hablará siempre de este). Dicha componente representa el espacio abstracto donde se renderizan las interfaces, las cuales serán objetos hijos del Canvas.

A la hora de implementar las diferentes UIs, en cada escena se han creado varios objetos hijo del *Canvas*, siendo cada uno una UI con un *tag* (etiqueta)

único que indique cuál es. De esta forma, se cambia de una interfaz a otra usando la función *FindGameObjectWithTag* de Unity, encontrando la UI y activando todo lo que contenga.

Cabe destacar que el objeto raíz de la UI no debe desactivarse puesto que de hacerse, no podrá encontrarse con su *tag*.

## 6.6. Combate

### 6.6.1. Hechizos

Para diseñar los hechizos, se ha usado la clase *ScriptableObject* de Unity. Esta clase de Unity crea contenedores de datos que permiten ahorrar mucho espacio ya que, si varios objetos tienen referencia a un *ScriptableObject*, no se crea un duplicado si no que se referencia al mismo. En este caso donde todos los jugadores tienen los mismos hechizos, conseguimos optimizar espacio.

Además, con el atributo *CreateAssetMenu* se facilita mucho el crear nuevos hechizos de cara al futuro, ya que, al incluirlos como opción a crear desde el editor, permite crearlos con un click de ratón.

A la hora de instanciar hechizos, se hace uso de un *prefab*, que es un *asset* de Unity que almacena un *GameObject* que sirve a modo de plantilla para todas las instancias que se hagan de él [29].

Para el *prefab* del hechizo, se ha dividido en dos scripts todo lo referente a este, uno siendo *SpellBehaviour*, que se encarga de su comportamiento; y el otro *SpellDisplay*, que se encarga de su aspecto.

### 6.6.2. Instanciamiento de jugadores

Para instanciar los jugadores en un combate, se usa la clase *PlayerSpawnSystem*, cuya función *SpawnPlayer* está suscrita al evento *OnServerReadied*, llamado cuando un cliente ha cargado la escena de combate. Un evento de Unity o *UnityEvent* es una herramienta que permite cambiar la devolución de una llamada de forma simple, añadiendo funciones a dicho evento que se ejecutarán cuando este sea invocado.

De esta forma, cuando un cliente cargue dicha escena, *PlayerSpawnSystem* se encargará de instanciar a su jugador y lo asignará a su conexión con la función *AddPlayerForConnection* de *Mirror*.

Además, hay varios puntos de *spawn* (instanciación) posibles que se recorren para instanciar cada jugador en un lugar distinto. Estos puntos son instancias de la clase *PlayerSpawnPoint*, dentro de la cual se declara la función *OnDrawGizmos*, lo que hará que dichos puntos se vean de una forma más clara en el editor, permitiendo editarlos mucho más fácilmente.

### 6.6.3. Sincronización entre jugadores

Dentro del combate, es necesario que todos los jugadores observen el juego en el mismo estado que el servidor.

En el caso de la posición de cada jugador, se usa la componente NetworkTransform de Mirror, la cual sincroniza cada cierto intervalo de tiempo posición, rotación y escala de cada jugador con el resto. Además, hace uso de interpolación por *snapshot*, explicada en el capítulo anterior, para que dicha sincronización se realice de forma suave y sea vista bien en pantalla.

En el caso de los hechizos, se utiliza otra componente, NetworkRigidbody2D, que realiza una función similar a NetworkTransform. Para tener más control sobre estos, se instancian con la función *NetworkServer.Spawn*, que, desde el servidor, instancia el hechizo en todos los clientes, donde solo importa su velocidad y dirección (controlados por NetworkRigidbody2D). Además, el uso de esta función facilita mucho la eliminación de hechizos cuando sea necesario, puesto que se realiza mediante la función *NetworkServer.Destroy*. El aspecto visual se sincroniza a través de un ClientRPC, un atributo que permite ejecutar una función en todos los clientes, usando un string a forma de código de color.

Para el color de los jugadores, se usan las herramientas Command, otro atributo que permite ejecutar una función en el servidor, y ClientRPC, explicada anteriormente. En este caso, desde cada cliente se envía un Command para que en el resto se cambie el color del jugador al correcto mediante un ClientRPC.

A su vez, también se usa ClientRPC cuando un jugador es derrotado para que se desactive su GameObject en todos los clientes, de forma que no interfiera en el resto de la partida.

## 6.7. Comunicación entre las entidades

### 6.7.1. Tipo de transporte

Mirror ofrece varios tipos de transporte a usar, sin embargo, se ha elegido KCPTTransport para este proyecto. Las razones principales para elegir KCPTTransport son su portabilidad (funciona en todas las plataformas excepto WebGL), que abre la posibilidad de usar el proyecto fuera de Linux y su velocidad (30 % - 40 % más rápido que TCP), que es de mucha importancia ya que los combates son de cierta rapidez. [30]

### 6.7.2. Mensajes

Con Mirror, se pueden declarar tipos de mensajes personalizados fácilmente, declarando un struct que herede de la clase NetworkMessage. Este struct puede mandar cualquier tipo básico.

Para mandarlos, Mirror ofrece opciones para enviar desde el cliente al server al que esté conectado (usando la función *NetworkClient.Send*) y del server a los clientes, donde se puede enviar a un solo cliente (con la función *Send* de *NetworkConnection*) o a varios (con *NetworkServer.SendToAll*, *SendToReady*, etc).

Para recibir un mensaje, se debe declarar una función que procese un tipo de mensaje en específico, usando *NetworkServer.RegisterHandler* o *NetworkClient.RegisterHandler* según sea conveniente.

## 6.8. Despliegue de la aplicación en Amazon Web Services

Para desplegar los servidores de la aplicación online, se ha hecho uso de los servicios proporcionados por Amazon Web Services (AWS), donde se puede optar a una versión gratuita durante un año, renunciando a espacio y potencia. Sin embargo, para este proyecto es suficiente y resulta una buena opción.

### 6.8.1. Configuración de máquina virtual

A la hora de comunicarse con el servidor alojado en la máquina virtual, es necesario configurar su *firewall* para que acepte el tráfico de red deseado. En este caso, al usar KCPTransport, basado en UDP, se permite que llegue tráfico UDP de cualquier dirección IP. Un ejemplo se muestra en la figura 6.1.

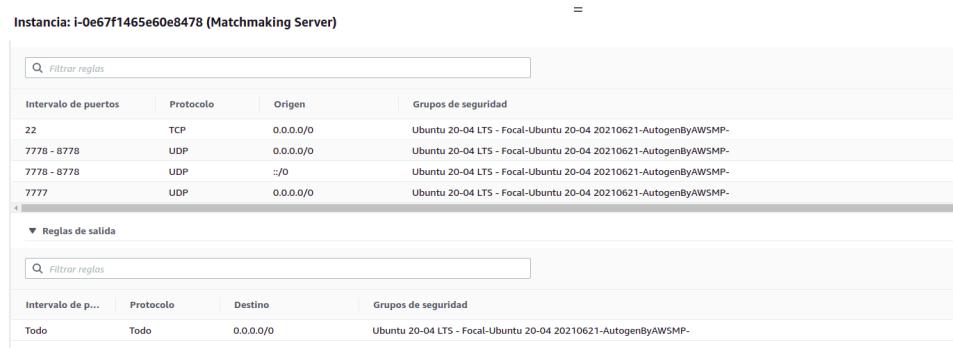


Figura 6.1: Ejemplo de reglas de seguridad del firewall de AWS

### 6.8.2. Despliegue del proyecto

Para poder realizar acciones de forma remota en la máquina virtual de AWS, se usan las herramientas proporcionadas por la *suite* OpenSSH, en específico, **sftp** y **ssh**.

La primera, **sftp** (*Secure File Transfer Protocol*) proporciona un canal seguro a través del cual intercambiar archivos con la máquina virtual. Por lo tanto, se usa para transferir los ejecutables de servidores y scripts necesarios para la ejecución del proyecto.

Por otro lado, **ssh** (*Secure SHell*) permite conectarse a la máquina virtual y operar en un shell sobre ella de forma segura, por lo que se usa para abrir el servidor de matchmaking una vez se haya transferido.

En este caso, para garantizar la seguridad a la hora de acceder al servidor, se proporciona una clave pública cuando se crea la máquina virtual que se usa a modo de identificación para ssh y sftp.



## **Capítulo 7**

# **Resultados experimentales**

En este capítulo se evalúa la tasa de refresco de la posición de los jugadores, necesario para minimizar el tráfico de red, y se comprueba el funcionamiento de la estructura propuesta usando máquinas virtuales.

### **7.1. Prueba sobre la tasa de refresco de componentes de sincronización**

En esta prueba se va a estudiar el efecto del intervalo de sincronización de la componente NetworkTransform explicada en el capítulo 6, para así poder ajustar su valor de forma que se genere menos tasa de bits debida a la sincronización, sin afectar a la calidad del juego. Nótese que cuanto más se reduzca la tasa de refresco o actualización, el movimiento, animación, etc. de los personajes del juego se verán menos fluidos.

#### **7.1.1. Entorno experimental**

Los detalles de la máquina usada para las pruebas se detallan en la figura 7.1

## 7.1. Prueba sobre la tasa de refresco de componentes de sincronización

Memoria	7,7 GiB
Procesador	Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4
Gráficos	NVIDIA Corporation GP107M [GeForce GTX 1050 Mobile] / GeForce GTX ...
Capacidad del disco	1,0 TB
Nombre del SO	Ubuntu 20.04.1 LTS
Tipo de SO	64 bits
Versión de GNOME	3.36.8
Sistema de ventanas	X11

Figura 7.1: Especificaciones de la máquina usada

El software usado es Unity 2019.4.12f1 junto a Mirror 35.1.0 y Wireshark 3.2.3 para la captura del tráfico generado.

### 7.1.2. Definición de la prueba

La forma en la que NetworkTransform funciona es que, cada cierto tiempo, envía datos de posición, rotación y escala del objeto con la componente al resto de clientes para que se sincronice. Ese cierto tiempo es a lo que llamamos intervalo de sincronización o tasa de refresco y es el valor que queremos ajustar.

Para esta prueba, se desactiva la interpolación por *snapshots* que usa NetworkTransform, puesto que así se puede observar y medir de forma más clara cómo funciona el juego con un determinado valor de sincronización, ya que se puede ver el momento exacto en el que el cliente sincroniza la posición del jugador.

La prueba consiste en, con una *build* del juego para un solo jugador, escoger un valor del intervalo de sincronización a probar, cambiarlo en las componentes que lo necesiten (NetworkTransform, NetworkAnimator, NetworkRigidbody), acceder a un combate donde solo estará un jugador, el cual se instancia en el borde izquierdo del mapa, y moverse de forma horizontal hasta llegar al límite derecho (es decir, de forma que todas las ejecuciones duren un tiempo similar).

Previo a entrar en el combate, se descubrirá el puerto usado por el jugador a través del comando de terminal `netstat -aun |grep 7779`, que devolverá las conexiones UDP en las que participe el puerto 7779, que usa el primer servidor de combate creado.

Dicha información se usará en la herramienta Wireshark, que captará el

tráfico cliente-servidor y servidor-cliente de la ejecución. Además, a través de un script en el objeto del jugador en Unity se escribirá en un archivo de texto los cambios en la componente x de la posición que ocurran. Ambos tipos de datos (Tráfico de red y cambios de posición) son los datos que se analizarán. Un ejemplo de la herramienta Wireshark se incluye en la figura 7.2.

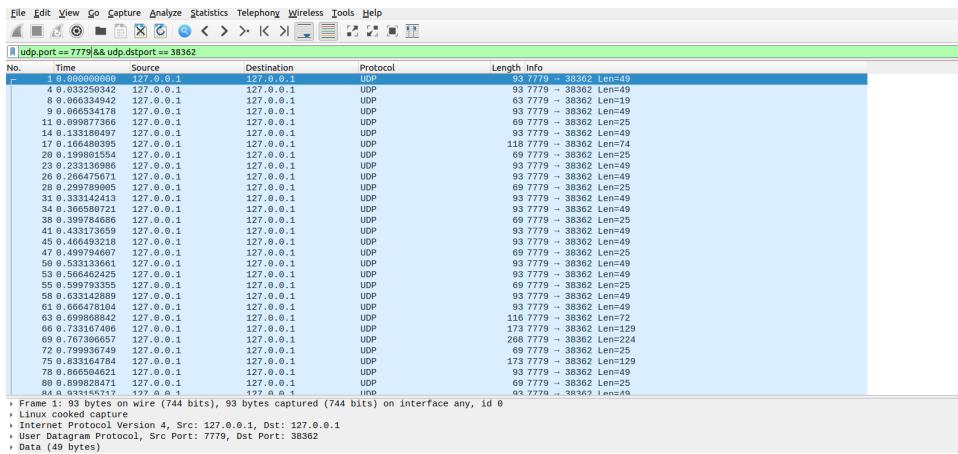


Figura 7.2: Ejemplo de uso de Wireshark

### 7.1.3. Parámetros de la prueba

Los intervalos de sincronización comprobados estaban comprendidos entre 0,1 y 0,2 segundos. La razón tras esto es que, por defecto, Mirror usa de intervalo de sincronización 0,1 s y funciona satisfactoriamente; y mediante pruebas, se comprobó visualmente que un intervalo más alto de 0,2 s es inviable a la hora de poder jugar. Así, los intervalos con los que se trabajaron fueron 0,1; 0,125; 0,15; 0,175 y 0,2 s.

Se mide el promedio en cambio de posición cada vez que esta se actualiza según la tasa de refresco. A partir de estos datos también se calculará la desviación típica a partir de la muestra obtenida y se estimará el error estándar de la media, es decir, cómo varían los resultados obtenidos del promedio, lo que nos indica cómo de constante es el tráfico de red. Viene calculado por la siguiente fórmula:

$$Error_{estandar} = \frac{DesviacionTipica}{\sqrt{Num.Iteraciones}} \quad (7.1)$$

Además, se medirá el bitrate generado por segundo, calculado por:

$$bitrate = \frac{\sum tamanoPaquetes}{Tiempofinal - Tiempoinicio} \quad (7.2)$$

#### 7.1.4. Resultados experimentales

Como se ha explicado en la sección de definición de la prueba, se analizan la magnitud en los cambios de posición del jugador y el tráfico generado con cada intervalo de sincronización probado.

##### Movimiento del jugador

En cuanto al movimiento horizontal, obtenido según el procedimiento detallado en la sección 7.1.3, se observan los siguientes promedios obtenidos en el cuadro 7.1. El error estándar, usado varias veces en este capítulo,

Movimiento horizontal			
Intervalo de sincronización (s)	Media de cambio en x	Desviación Típica media	Error estándar de la media
0,100	1,034	0,330	0,125
0,125	1,195	0,349	0,132
0,150	1,459	0,460	0,174
0,175	1,733	0,568	0,214
0,200	1,905	0,508	0,192

Cuadro 7.1: Tabla de resultados de cambio de movimiento en x.

En la figura 7.3 se observa cómo incrementan el tamaño medio de los "saltos" que da el jugador a la vez que las sincronizaciones del servidor son más distantes entre sí. En cuanto al error estándar, representado en la figura por los segmentos negros en cada punto, se debe a que la velocidad no es constante, a la vez que el tiempo de llegada de los paquetes, por lo que los saltos tampoco serán siempre de la misma longitud. Por dar algo más de contexto a los datos obtenidos, de forma visual, dichos saltos comienzan a notarse significativamente a partir de 0,15 segundos de intervalo a la hora de realizar acciones.

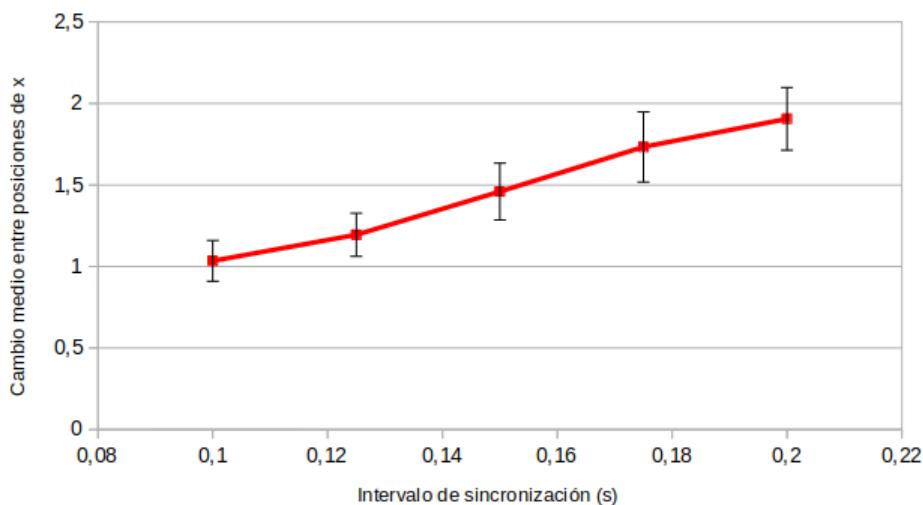


Figura 7.3: Gráfica de cambio medio en la x del jugador con respecto al intervalo de sincronización

#### Tráfico desde el cliente al servidor

En el cuadro 7.2 se observan los datos obtenidos, siendo las medias a partir de 7 ejecuciones por cada valor de intervalo probado.

Tráfico generado (Cliente-Servidor)			
Intervalo de sincronización (s)	Bitrate generado medio (b/s)	Desviación Típica media (b/s)	Error estándar
0,050	45438,075	—	—
0,100	42298,902	410,732	155,242
0,125	41906,320	384,789	145,436
0,150	41622,383	331,549	125,314
0,175	41545,137	702,602	265,559
0,200	41390,408	581,153	219,655

Cuadro 7.2: Tabla de tráfico generado desde el cliente al servidor.

Representados gráficamente, en la figura 7.4, primero se observó que el tráfico generado trazaba una curva descendente según aumentaba el valor del intervalo, similar a una función racional, habiendo además muy poca diferencia con la media. Para tener una idea más clara de la forma que sigue el tráfico generado, se realizaron también pruebas con el valor 0,05; como se puede observar, se acentúa la similitud con una función racional ya que aumenta enormemente al llegar a valores pequeños.

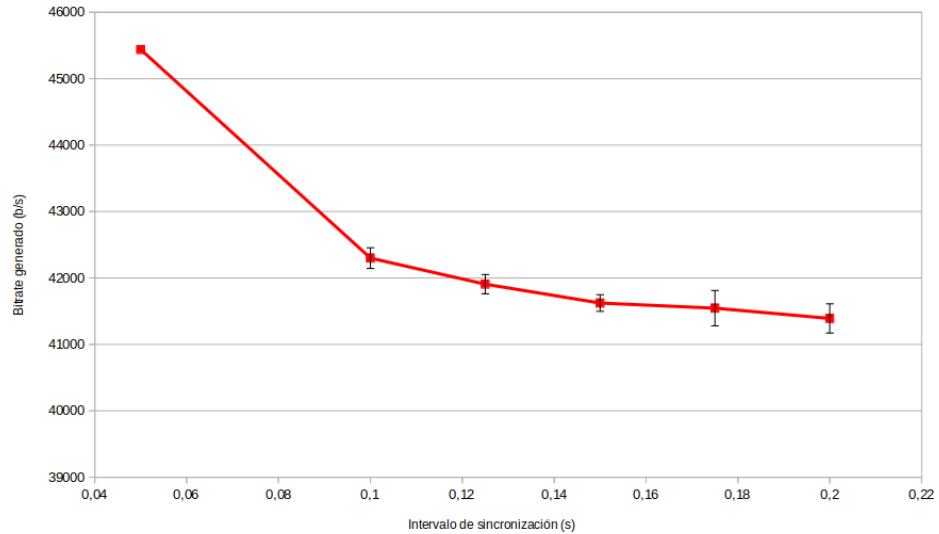


Figura 7.4: Gráfica de media de bitrate (Cliente a servidor)

Para poder encontrar alguna relación más clara entre el tráfico generado y el intervalo escogido, también se midió el número de paquetes capturados por Wireshark, calculando el número de paquetes generados por segundo para tres intervalos equidistantes. En el cuadro 7.3 se observan los datos recogidos:

Paquetes generados (Cliente-Servidor)			
Intervalo de sincronización (s)	Paquetes/s	Desviación Típica media	Error estándar de la media
0,100	54,822	1,216	0,459
0,150	54,065	0,837	0,316
0,200	53,690	1,675	0,633

Cuadro 7.3: Tabla de paquetes generados (Cliente a servidor).

Representando gráficamente el número de paquetes por segundos en la figura 7.5, se observa un comportamiento cercano de imitar al obtenido en la figura 7.4, aunque harían falta más valores pequeños para ver si también crece de la misma forma, ya que la diferencia entre los valores obtenidos no es mayor de 2 paquetes por segundo.

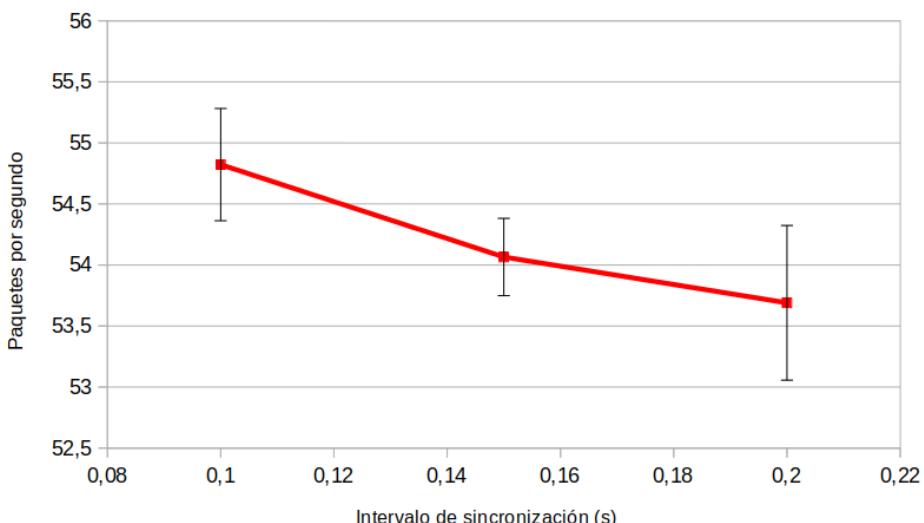


Figura 7.5: Gráfica de paquetes generados por segundo (Cliente a servidor)

#### Tráfico desde el servidor al cliente

En el tráfico servidor-cliente, los datos recogidos son los mostrados en el cuadro 7.4, que surgen de las mismas 7 ejecuciones que en el apartado anterior.

Tráfico generado (Servidor-Cliente)			
Intervalo de sincronización (s)	Bitrate generado medio (b/s)	Desviación Típica media	Error estándar de la media
0,050	30681,371	—	—
0,100	25444,481	325,860	123,164
0,125	25170,277	197,095	74,495
0,150	24350,014	144,531	54,627
0,175	24078,653	193,631	73,186
0,200	23986,358	196,241	74,172

Cuadro 7.4: Tabla de tráfico generado (Servidor a cliente).

En la figura 7.6 se observa un comportamiento similar al tráfico entre cliente y servidor, sin embargo, desde el servidor hay menos variación entre intervalos de sincronización distintos exceptuando por valores pequeños, donde en ambos casos se dispara. Al contrario que en figuras anteriores, no se especifica el error estándar puesto que es tan pequeño respecto a la escala de la gráfica que no es observable (de media es un 0,3 % del bitrate aproximadamente).

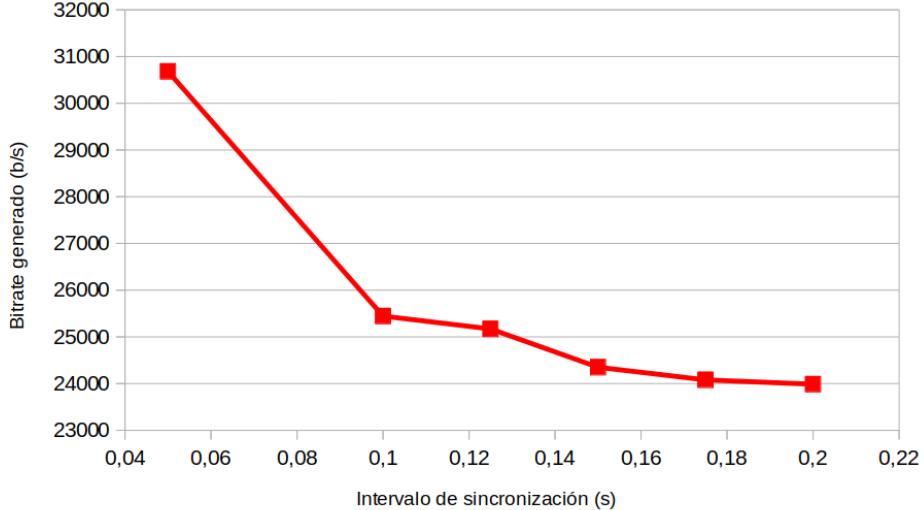


Figura 7.6: Gráfica de media de bitrate generado (Servidor a cliente)

En el caso de los paquetes generados, los datos obtenidos son los mostrados en el cuadro 7.5:

Paquetes generados (Servidor-Cliente)			
Intervalo de sincronización (s)	Paquetes/s	Desviación Típica media	Error estándar de la media
0,100	30,695	0,086	0,032
0,150	30,713	0,107	0,040
0,200	30,677	0,107	0,040

Cuadro 7.5: Tabla de paquetes generados desde el servidor al cliente.

En la figura 7.7 se muestra que se produce una variación del número de paquetes por segundo prácticamente despreciable (inferior a 0,025 paquetes/s).

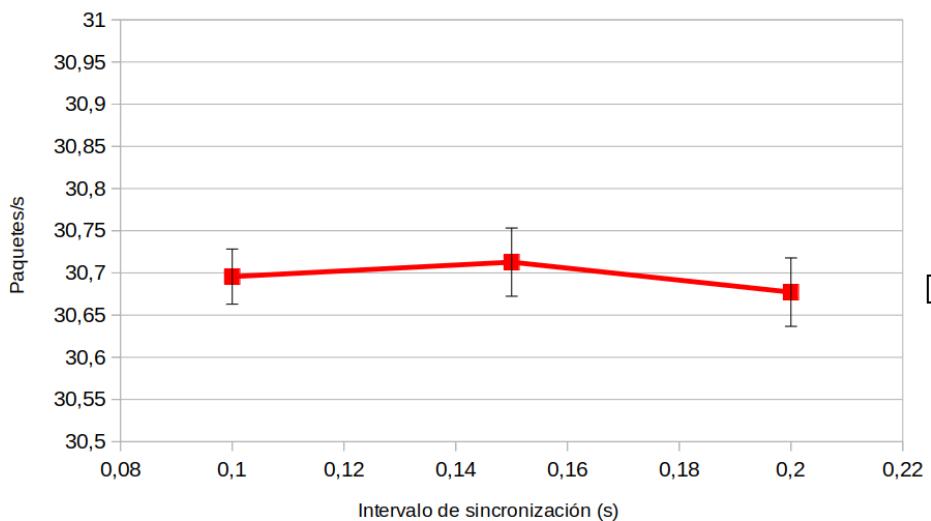


Figura 7.7: Gráfica de paquetes generados por segundo (Servidor a cliente)

Vemos que viendo el tráfico generado desde el servidor al cliente, no hay mucha diferencia entre los valores 0,1 s y 0,2 s. Sin embargo, las diferencias se acentúan en el otro sentido. Finalmente, a la hora de jugar, vemos un crecimiento lineal en la distancia recorrida entre actualizaciones según el intervalo usado. Por lo tanto, estos dos últimos factores serán lo más importantes a la hora de decidir qué tasa de refresco usar para el juego.

Si se valora más minimizar el tráfico generado, usar un valor en torno a 0,15 s puede ser recomendable ya que un mayor valor produce saltos demasiado grande que afectarán de forma negativa a la experiencia de jugador. En el caso de poder utilizar mayor ancho de banda, la tasa de 0,1 s es lo más recomendable puesto que se asegura la satisfacción del jugador. Finalmente, en 0,125 s se encuentra un punto intermedio entre ambos fines, pero no por ello es el más recomendado puesto que depende del contexto de red en el que nos encontremos.

## 7.2. Prueba de instanciación remota de servidores para escalado

En esta prueba se va a medir los recursos computacionales necesarios para el funcionamiento de los distintos servidores, y estudiar el funcionamiento de la estructura de servidores distribuidos detallada en el capítulo 5 mediante máquinas virtuales.

## **9.6.2. Prueba de instanciación remota de servidores para escalado**

### **7.2.1. Entorno**

Además del entorno descrito en la subsección 7.1.1, se usa VirtualBox 6.1 para lanzar distintas máquinas virtuales, las cuales usan Lubuntu 20.04.3 como sistema operativo, poseen 1GB de memoria RAM y 1 CPU cada una. Además, se usan las herramientas *atop* y *ps* para medir el porcentaje de cpu y memoria que se consume.

### **7.2.2. Definición de la prueba**

En esta prueba se comprobará si es posible el funcionamiento de la estructura de servidores propuesta en el capítulo 5 haciendo uso de varias máquinas virtuales ajenas a la máquina principal. Recuérdese que esta distribución de servidores por funciones permitiría lanzar servidores según el número de partidas y jugadores. Además, se medirá la carga consumida en CPU y memoria de cada tipo de servidor. De esta manera, se podrá tener un criterio claro sobre en qué máquina se podría lanzar un servidor de partida o combate, conociendo qué recursos disponibles tiene, y qué recursos consume cada servidor.

Para comprobar la carga consumida, se usará principalmente el comando *ps -o %cpu, %mem, command ax |grep \$NOMBRE\_EJECUTABLE\_SERVIDOR*, que proporcionará la carga de CPU y memoria RAM que está usando el proceso del servidor. En caso de necesitar información más específica, se reunieron también datos con el comando *atop*, cuya salida proporciona información más en profundidad sobre el rendimiento del sistema.

### **7.2.3. Parámetros de la prueba**

Como parámetros a medir en esta prueba, se tendrán en cuenta el porcentaje de CPU y la cantidad de memoria en MB que se usa para cada tipo de servidor.

### **7.2.4. Resultados experimentales**

Se separan los datos según se haya hecho todo en local y usando máquinas virtuales:

#### **En local**

En el cuadro 7.6 se muestran los datos obtenidos separados según el tipo de servidor. Siendo en local, se realiza con 1 y 4 jugadores para comprobar la carga que va a tener que soportar los servidores.

Como se observa en la figura 7.8, la carga aumenta según el tipo de servidor. Estos resultados son coherentes: el servidor de matchmaking solo alberga los datos de los jugadores y realiza alguna operación momentánea

Carga CPU (Local) (% de uso)			
Num. Jugadores	Matchmaking	Partida	Combate
1	1,12	1,36	1,88
4	1,1	1,48	2,36

Cuadro 7.6: Tabla de carga de CPU lanzando todo en local

(comunicación con MongoDB y creación de server de partida), el servidor de partida tiene una carga similar a este, pero su funcionamiento depende más del número de jugadores que el servidor de matchmaking, y el de combate es el que más carga soporta con diferencia puesto que se encarga de la instanciación de objetos, sincronización y comunicación de jugadores durante el combate, lo que aumenta con 4 jugadores ya que los combates albergan dos jugadores en vez de solo uno.

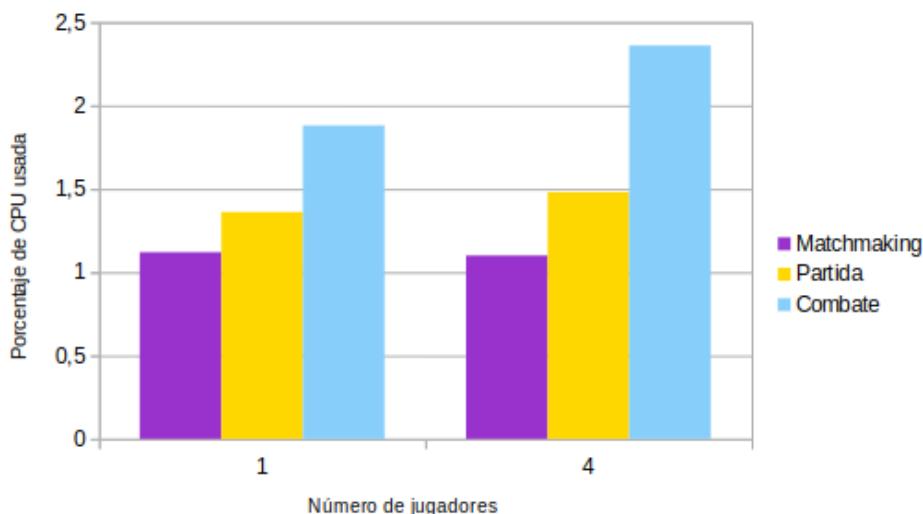


Figura 7.8: Gráfica de comparación de la carga CPU lanzando todos los servidores en local.

En cuanto a la memoria, tenemos los siguientes datos en el cuadro 7.7:

Carga memoria (Local) (Megabytes)			
Num. Jugadores	Matchmaking	Partida	Combate
1	55,193	70,963	70,963
4	59,924	70,963	70,963

Cuadro 7.7: Tabla de carga de memoria lanzando todo en local

En la figura 7.9, vemos como la carga en memoria se mantiene constante en los servidores de partida y combate, mientras que crece para el servidor

## **9.8.2. Prueba de instancia remota de servidores para escalado**

---

de matchmaking.

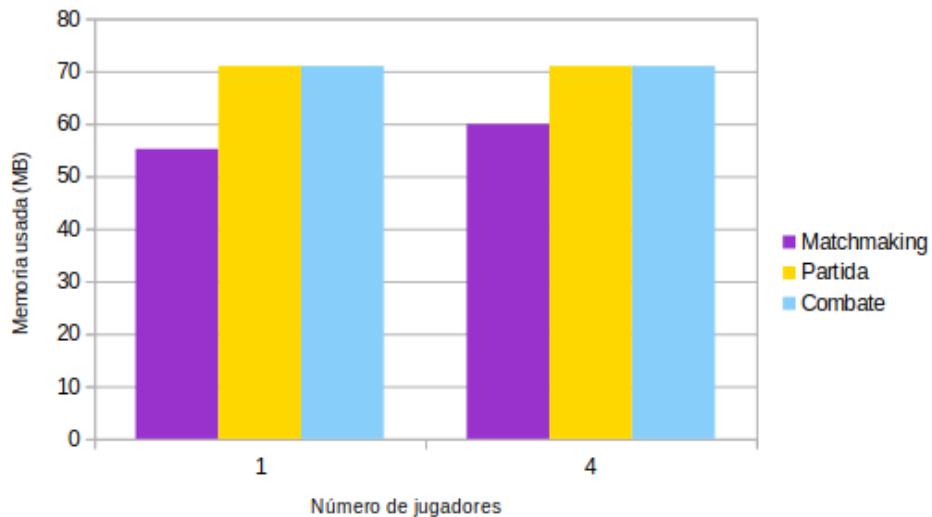


Figura 7.9: Gráfica de carga de memoria en local

Conociendo estos datos, se podría elaborar un algoritmo que permita al servidor de matchmaking principal equilibrar la carga, de forma que pueda distribuir la ejecución de servidores según la disponibilidad de memoria que posea cada uno.

### **Usando máquinas virtuales**

Haciendo uso de máquinas virtuales, se lanza cada tipo de servidor en una máquina distinta. Como las pruebas se realizan en uno o dos combates, se necesitarán como máximo tres máquinas virtuales, ya que el servidor de matchmaking se lanzará desde la máquina base, y luego se lanzará el servidor de partida y como máximo, dos servidores de combate (en el caso de cuatro jugadores). De esta manera se evaluará si el procedimiento para instanciar los servidores, y comunicar los datos de conexión a los otros participantes, es correcto.

Solo se han realizado las pruebas con un jugador, y los datos obtenidos son los que figuran en el cuadro 7.8.

Se observa una carga similar en cuanto a los servidores solo, pero sin embargo, el uso de máquinas virtuales añade mucha carga en memoria para la máquina base.

Para concluir, se ha comprobado el correcto funcionamiento de la instancia remota de servidores en distintas máquinas, que habilita la posibilidad de usar dicha característica para ampliar la escalabilidad del juego. Además, con los datos reunidos sobre la carga de memoria, se podría realizar un algo-

Carga memoria (Local) (Megabytes)				
Matchmaking	Partida (Servidor + VM)	Partida (Servidor)	Combate (Serv. + VM)	Combate (Serv.)
55,193	977,715	71,764	989,542	74,017

Cuadro 7.8: Tabla de carga de memoria lanzando servidores en máquinas virtuales

ritmo que equilibre la carga entre servidores, de forma que ninguno realice más esfuerzo computacional que el resto. En un futuro, esta información podría permitir decidir incluso instanciar bajo demanda una nueva máquina virtual, en los casos en que no hubiera suficientes recursos en el resto de máquinas. Concretamente, teniendo en cuenta los valores más altos de uso evaluados, se podrían lanzar servidores de partida o de combate en máquinas que tuvieran al menos 71MB de memoria RAM libre, y un porcentaje de uso de CPU inferior al 97%.



# Capítulo 8

## Conclusiones

### 8.1. Conclusiones

El sector de los videojuegos *online* sigue siendo un mercado interesante en el mundo de la tecnología, pero con una gran cantidad de jugadores surgen problemas relacionados con la escalabilidad del servidor usado para proporcionar a los jugadores su experiencia.

En este proyecto se ha realizado un juego multijugador de acción 2D. Con el fin de que sea escalable, se ha diseñado una estructura basada en tres tipos de servidores que se comunican entre ellos para manejar el flujo de las partidas y que permite lanzar servidores en diferentes máquinas.

Para realizarlo, se ha usado el motor de desarrollo de videojuegos Unity debido a sus facilidades para desarrolladores primerizos y su enfoque en juegos 2D. En cuanto al multijugador, se ha usado el framework Mirror para programar el funcionamiento del juego, y MongoDB junto a NodeJS para implementar la base de datos donde se almacenan los datos de los jugadores. El despliegue del juego se realiza usando Amazon Web Services (AWS), para que sea accesible a todos los jugadores en cualquier momento.

En cuanto a las pruebas realizadas, se ha observado el comportamiento del tráfico con respecto a la tasa de refresco del servidor hacia los clientes. Resulta interesante cómo explota el crecimiento del tráfico usando valores muy pequeños de dicha tasa. A su vez, se ha comprobado el funcionamiento que se planteó en su diseño haciendo uso de máquinas virtuales para simular el proceso de crear servidores a distancia y distribuir la carga de estos, viendo los requisitos de memoria necesarios para que los servidores lleven a cabo su función.

### 8.2. Trabajo futuro

En cuanto a cambios futuros en el proyecto, hay varios aspectos a mejorar para garantizar su robustez. Uno sería mejorar la transparencia de los

servidores con los clientes, para que se proporcione más información útil para el jugador sobre cada proceso.

Otro aspecto a mejorar es la seguridad en la comunicación con los servidores, donde se podría usar la componente de validez de los Json Web Tokens para mejorar la experiencia de usuario y aumentar la seguridad del sistema.

Viendo el proyecto como un videojuego, sufre mucho en el aspecto audiovisual, donde faltan muchos aspectos importantes para la experiencia del usuario: música, efectos de sonido, animaciones, etc.

También es necesario un diseño de matchmaking mejor, que tenga en cuenta la habilidad de cada jugador, puesto que el actual es muy débil y no garantiza una partida que se sienta satisfactoria para el jugador. Dicho cambio supondría también una ampliación de la base de datos para almacenar más datos sobre cómo se desenvuelve cada jugador.

## Apéndice A

# Instrucciones de uso

Al iniciar el juego, se observa una pantalla como la que se observa en la figura A.1. Se puede pulsar login si ya se posee una cuenta o registrarse en caso de necesitar crear una.

Al logearse en el juego, se buscará una partida, tras lo cual aparecerá una pantalla como la especificada en A.2.

Cuando todos los jugadores pulsen el botón de “COMENZAR”, se pasará a la pantalla de la figura A.3, donde, si todos los jugadores pulsan el botón disponible, pasarán al combate.

Dentro del combate, descrito en la figura A.4 se usan los siguientes controles:

- Teclas W, A, S, D para mover al jugador horizontal y verticalmente.
- Ratón para apuntar los hechizos.
- Rueda de ratón para seleccionar el hechizo a lanzar/marcar.
- Barra espaciadora para marcar un hechizo.
- Click derecho para lanzar un hechizo sencillo (el seleccionado).
- Click izquierdo para lanzar un hechizo combinado con los hechizos agrupados.

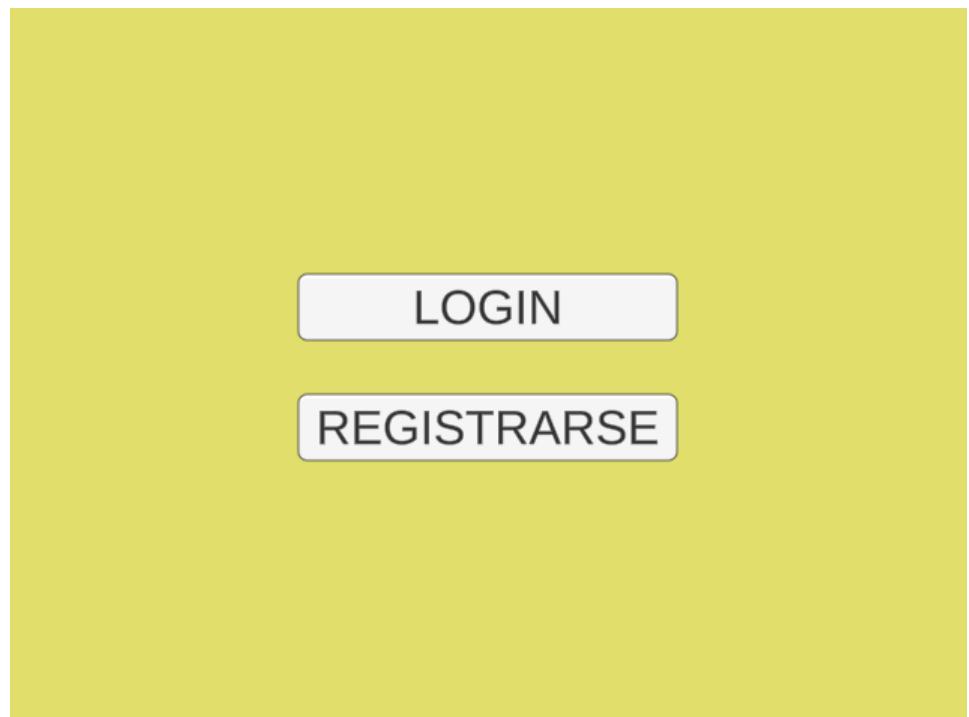


Figura A.1: Pantalla de menú

USUARIO	1er	2o	3o	TOTAL
u4	0	0	0	0
u3	0	0	0	0
u2	0	0	0	0
u1	0	0	0	0

**COMENZAR**

Figura A.2: Pantalla de partida



Figura A.3: Pantalla previa al combate



Figura A.4: Pantalla de combate



# Glosario

## A

### **AAA**

Sobre un videojuego, de desarrollo con un alto presupuesto.. 26

### **API**

Interfaz de Programación de Aplicaciones, que permite comunicar dos sistemas o plataformas diferentes.. 79, 80

### **atributo**

En Unity, marcador que pueden ser colocados sobre una clase, propiedad o función en un script indicando un comportamiento especial. En Unity se especifican entre corchetes encima del elemento que afecta.. 82, 83

### **AWS**

Amazon Web Services. 46, 101

## C

### **CCU**

Concurrently Connected Users (Jugadores conectados concurrentemente). 29

### **command**

Herramienta de Mirror que permite ejecutar, desde el cliente, una función en el servidor.. 39, 40

## F

### **framework**

entorno de trabajo o estructura previa que ayuda a resolver un problema de programación.. 13, 19, 24–29, 46, 51, 101

**I****indie**

Independiente, sin el apoyo de una distribuidora.. 24, 25

**L****LTS**

Referido a una versión de un producto, significa Long Term Service, es decir, versión de un producto diseñada para tener soporte a largo plazo.. 27

**M****matchmaking**

Matchmaking o emparejamiento es el proceso mediante el cual se unen a varios jugadores en una misma sesión de juego mediante algún criterio.. 14, 17, 49–52, 59–62, 64, 70

**P****pixel art**

Forma de arte digital donde las imágenes se tratan a nivel de pixel. 24

**R****roguelike**

Género de videojuego definido por, entre otras características, muerte permanente del jugador (si es derrotado ha de empezar desde el principio), contenido generado aleatoriamente, etc.. 23

**S****script**

Una fragmento de código que dice a un objeto cómo ha de comportarse. En Unity, te permite crear componentes propios, responder al input del usuario, etc.. 14, 80, 82, 89

**shell**

Término de UNIX que se refiere a la interfaz interactiva del usuario con el sistema operativo.. 85

**shoot’em up**

Género de videojuego en el que el objetivo es que un solo personaje elimine a cuantos más enemigos posibles.. 23

**sprite**

Mapa de bits dibujado en la pantalla del ordenador. 25, 45, 46

**struct**

Estructura compuesta de datos de otros tipos.. 83

**T****token**

Objeto o identificador que representa el derecho de una entidad a ejecutar una acción.. 50, 51, 53, 59, 61, 62

**U****UI**

User Interface (Interfaz de usuario). 36, 39, 40, 44, 47







# Bibliografía

- [1] DEV. *Libro Blanco del Desarrollo Español de Videojuegos*. URL: <https://dev.org/images/stories/docs/libro%20blanco%20del%20desarrollo%20espanol%20de%20videojuegos%202020.pdf>.
- [2] Marc Brugat. *Fortnite generó más de 9.200 millones de dólares entre 2018 y 2019*. URL: <https://www.lavanguardia.com/videojuegos/20210504/7426987/fortnite-genero-mas-9-200-millones-dolares-2018-2019-ventas.html>.
- [3] Stefan Hall. *COVID-19 is taking gaming and esports to the next level*. URL: <https://www.weforum.org/agenda/2020/05/covid-19-taking-gaming-and-esports-next-level/>.
- [4] Nick Statt. *Fortnite is now one of the biggest games ever with 350 million players - The Verge*. URL: <https://www.theverge.com/2020/5/6/21249497/fortnite-350-million-registered-players-hours-played-april>.
- [5] Daniel Escandell. *Análisis de Lost Magic*. 2006. URL: <https://vandal.elespanol.com/analisis/nds/lost-magic/5265#p-29>.
- [6] *Página en Steam de Nuclear Throne*. 2021. URL: [https://store.steampowered.com/app/242680/Nuclear\\_Throne/](https://store.steampowered.com/app/242680/Nuclear_Throne/).
- [7] *Página en Steam de Duck Game*. 2021. URL: [https://store.steampowered.com/app/312530/Duck\\_Game/](https://store.steampowered.com/app/312530/Duck_Game/).
- [8] Rafael Valencia-Garcia y col. *Technologies and Innovation: Second International Conference, CITI 2016, Guayaquil, Ecuador, November 23-25, 2016, Proceedings*. Springer International Publishing, 2016. ISBN: 9783319480244.
- [9] *Creador y editor de juegos 2D y 3D — Software de realidad aumentada/virtual — Motor de juegos — Unity*. 2021. URL: <https://unity.com/es/products/unity-platform>.
- [10] *Mirror Networking - Open Source Networking for Unity*. 2021. URL: <https://mirror-networking.com/>.
- [11] *Photon Unity 3D Networking Framework SDKs and Game Backend*. 2021. URL: <https://www.photonengine.com/pun>.

- [12] *Crea fácilmente videojuegos con GameMaker Studio 2.* 2021. URL: <https://www.yoyogames.com/es/gamemaker>.
- [13] *Videojuegos hechos con GameMaker.* 2021. URL: <https://www.yoyogames.com/es/showcase>.
- [14] *Game Engine — Build Multi-Platform Video Games - Unreal Engine.* 2021. URL: <https://www.unrealengine.com/en-US/solutions/games>.
- [15] Unity Technologies. *Unity - Manual: Multiplayer and Networking.* 2019. URL: <https://docs.unity3d.com/Manual/UNet.html>.
- [16] *UNet Deprecation FAQ.* 2019. URL: <https://support.unity.com/hc/en-us/articles/360001252086-UNet-Degradation-FAQ>.
- [17] *RPCs and RaiseEvent — Photon Engine.* URL: <https://doc.photonengine.com/en-us/pun/current/gameplay/rpcsandraiseevent>.
- [18] Ian Sommerville. *Ingeniería de software (9th ed.)* Pearson Educación, 2011. ISBN: 978-607-32-0603-7.
- [19] Roger S. Pressman. *Ingeniería de software. Un enfoque práctico.* McGraw-Hill, 2011. ISBN: 978-607-15-0314-5.
- [20] *Scrum.org.* 2021. URL: <https://www.scrum.org/>.
- [21] Ken Schwaber y Jeff Sutherland. *La Guía Scrum.* Traducido del original en inglés por David Hernán Tardini y David Martí. 2020. URL: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Spanish-European.pdf>.
- [22] RGS\_Dev. *Animated Mages Character Pack by RGSDev.* URL: <https://rgsdev.itch.io/pixel-art-animated-mages-character-pack-rgsdev>.
- [23] PixelPoem. *2D Pixel Dungeon Asset Pack by Pixel\_Poem.* URL: <https://pixel-poem.itch.io/dungeon-assetpack>.
- [24] *Salario de un Programador/a junior en España.* Se actualiza en tiempo real, por lo que el valor usado en este documento puede variar con respecto a la página. 2021. URL: <https://es.indeed.com/career/programador-junior/salaries>.
- [25] OpenJS Foundation. *La base de datos líder del mercado para aplicaciones modernas — MongoDB.* URL: <https://www.mongodb.com/es>.
- [26] Inc. MongoDB. *Node.js.* URL: <https://nodejs.org/es/>.
- [27] TJ Holowaychuk, StrongLoop y col. *Express - Infraestructura de aplicaciones web Node.js.* URL: <https://expressjs.com/es/>.
- [28] Alexander Batishchev. *Jwt.Net, a JWT (JSON Web Token) implementation for .NET.* URL: <https://github.com/jwt-dotnet/jwt#jwtnet-a-jwt-json-web-token-implementation-for-net>.

- [29] Unity Technologies. *Unity - Manual: Prefabs*. 2016. URL: <https://docs.unity3d.com/es/530/Manual/Prefabs.html>.
- [30] Lin Wei. *KCP - A Fast and Reliable ARQ Protocol*. URL: <https://github.com/skywind3000/kcp/blob/master/README.en.md#kcp---a-fast-and-reliable-arq-protocol>.