UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

20|21

Trabajo fin de grado

Implementación de un servidor para juegos RPG online multijugador

Arturo Morcillo Penares

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería informática

TRABAJO FIN DE GRADO

Implementación de un servidor para juegos RPG online multijugador

Autor: Arturo Morcillo Penares
Tutor: Carlos Aguirre Maeso

junio 2021

Arturo Morcillo Penares
Implementación de un servidor para juegos RPG online multijugador

Arturo Morcillo Penares
C\ Pedro Jiménez nº15

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

La desconfianza es la madre de la seguridad.

Aristófanes

Agradecimientos

Me gustaría agradecerle a mi madre por todo lo que me ha ayudado y todo el trabajo que ha hecho desde que era pequeño hasta que he llegado aquí.

Resumen

El objeto de este trabajo de fin de grado es desarrollar un servidor para videojuegos de rol en vista isométrica con un sistema de seguridad que garantice que el jugador no hará trampa.

Este servidor tendrá un esquema autoritario que no deje al cliente proporcionar ningún tipo de información del estado del juego, solo podrá enviar sus acciones como hacer click, pulsar una tecla... Del mismo modo, se garantizará un correcto cifrado de la información sensible para evitar que sea obtenida por parte de terceros.

La información de los usuarios se almacenará en una base de datos. Esta base de datos cifrará la información sensible y se llevarán a cabo las medidas necesarias para evitar que cualquier tercero pueda acceder a la información de esta base de datos.

Este trabajo pretende servir como base de un servidor para juegos de acción multijugador. Permitirá que otros desarrolladores lo puedan utilizar para desarrollar sus propios videojuegos de acción en tiempo real o juegos de rol multijugador masivo.

También se incluirá un videojuego de ejemplo para mostrar el funcionamiento del servidor.

Palabras clave

Servidor, Ciberseguridad, base de datos, inyección SQL, cifrado, clave pública, clave privada, pentesting

Abstract

The purpose of this document is the development of an isometric rol videogame server with a security system that doesn't allow cheating.

This server will have an authoritative scheme that doesn't allow client to supply any information about game state. Client only sends inputs as clicks, key pressing... In the same way, the server cypher sensitive data in order avoid external people to read it.

User data will be stored in a database. Database will cypher sensitive data and the necessary measures will be taken to avoid any people to obtain database information.

This work pretends to be an open source server to develop multiplayer action games. It will be available to other developers to help them to develop their own real time action games or massive multiplayer games.

An example videogame will be included in order to show the correct performance of the server.

Keywords

Server, cybersecurity, database, sql injection, Cypher, public key, private key, pentesting

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de cuadros

Lista de ecuaciones

Lista de figuras

Lista de tablas

Lista de cuadros

1

Introducción

El objeto de este trabajo es desarrollar un servidor de videojuegos que posea un sistema de seguridad para evitar las trampas por parte de otros usuarios.

1.1. Motivación

Durante los años pasados la popularidad de los videojuegos online creció de manera notoria. Este hecho se vió potenciado por la pandemia del 2020 ya que entre los juegos más jugados de este año se encuentran Among us, League of Legends, Fornite y Fall Guys entre muchos otros [1].

Conforme cualquier medio gana en popularidad se aprecia un aumento en la cantidad de ataques y maneras de aprovecharse de cualquier fallo de seguridad.

En los videojuegos de un solo jugador el hecho de hacer trampas no es un factor relevante ya que solo afecta a la experiencia del usuario que está haciendo esas trampas. No obstante, esto cambia en los juegos multijugador online. En este medio un jugador haciendo trampas puede afectar a decenas, cientos o incluso miles de jugadores.

Crear un servidor de videojuegos no es una tarea especialmente complicada, pero cuando buscamos una mayor seguridad se puede volver una tarea más laboriosa. Muchos desarrolladores se enfrentan a los propios jugadores haciendo trampas y a terceras personas intentando alterar el estado del juego de manera ilegal. En los juegos tácticos que van por turnos se puede realizar un control de los paquetes más minucioso, pero en juegos más orientados a la acción resulta complicado.

La finalidad de este trabajo consiste en ofrecer un ejemplo de servidor centrado en los juegos de acción y con un sistema de seguridad robusto. Para probar su correcto funcionamiento se desarrollará un videojuego que lo utilice y se llevará a cabo un proceso de pentesting.

1.2. Objetivos

El objetivo de este proyecto es desarrollar un sistema que permita jugar a un videojuego RPG de acción entre varios jugadores evitando trampas por parte del usuario. Para esto tendremos:

1.– Un servidor que permita autenticarse, controle y simule el estado del juego.

2.– Una base de datos con la información de los jugadores registrados.

3.– Un cliente que muestre el estado del juego para el jugador y se comunique con el servidor.

4.– Un protocolo de comunicación con el servidor que dificulte las trampas por parte del usuario.

5.– Un juego de rol de acción estilo *diablo* [2].

1.3. Estructura

Este documento está dividido en 7 apartados:

- **Introducción:** Apartado en el que nos encontramos. Supone un punto de partida para explicar este trabajo
- **Estado del arte:** Se lleva a cabo un análisis de las tecnologías y arquitecturas más utilizadas para videojuegos online, dando ejemplos que las utilicen. Se analizarán pros y contras de cada tecnología de forma crítica.
- **Definición:** Se definirá el alcance del proyecto. La parte más importante de este punto es el análisis de requisitos funcionales y no funcionales.
- **Diseño:** Se plantea el diseño de la aplicación y los protocolos empleados para comunicarnos con el servidor. En este punto se tratarán el manejo de la información y como esta es sincronizada.
- **Desarrollo:** Se explica el proceso llevado a cabo para desarrollar el trabajo. Se tratan las iteraciones realizadas, cambios realizados en ellas hasta el diseño final y una pequeña exposición del resultado final.
- **Pruebas:** Se detallan las pruebas realizadas. Tanto las pruebas a las funciones individuales, al manejo de paquetes y el pentesting.
- **Resultado y trabajo futuro:** Se resume el resultado del proyecto y se explican campos donde aplicar esta tecnología.

2

Estado del Arte

En el mundo de los videojuegos online hay que tomar varias decisiones. Entre estas se encuentran el protocolo de comunicación, la arquitectura del sistema y las distintas medidas de seguridad que se emplearán.

Actualmente existen varias posibilidades para establecer la interacción entre los distintos jugadores. En este trabajo, se actualiza el estado mediante el protocolo UDP y con una arquitectura cliente-servidor.

El principal motivo de utilizar una estructura cliente-servidor es debido a las ventajas que ofrece en el ámbito de la seguridad. También permite mantener la persistencia de forma mucho más sencilla y efectiva. De hecho, en casi todos los videojuegos multijugador online se emplea este sistema.

El sistema presentado se basa en el tutorial creado por Tom Weiland [3], implementando protocolos de comunicación y manejo de la información que eviten modificaciones de usuarios externos y de los propios usuarios.

En este apartado se detallan las prácticas y tecnologías más comunes en el ámbito de los videojuegos online para justificar estas elecciones.

2.1. Arquitectura

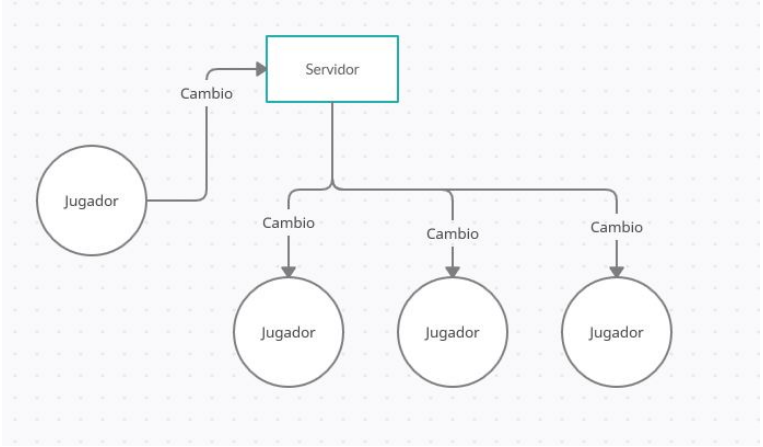
Actualmente tenemos tres arquitecturas sin contar esquemas híbridos:

- Servidor dedicado
- Listen server
- Peer to Peer

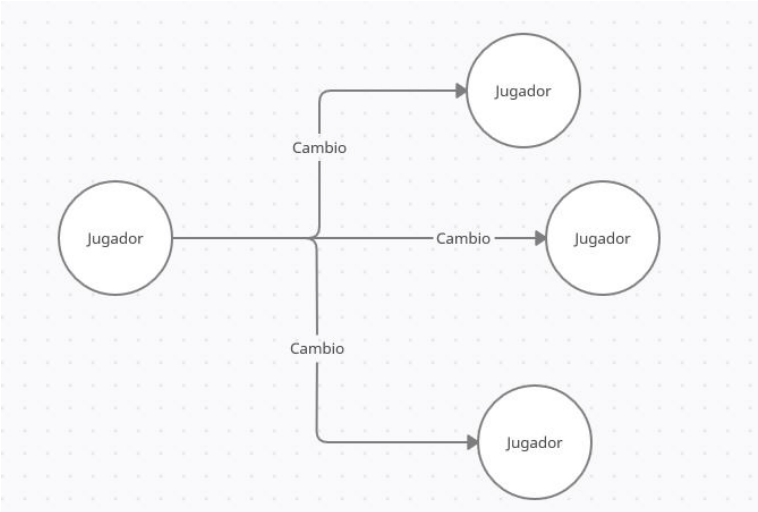
Lo primero que tenemos que tener en cuenta es que un servidor dedicado y un listen server se basan en un esquema cliente-servidor.

En una arquitectura cliente-servidor el cambio es notificado al servidor. Acto seguido se notifica el cambio al resto de jugadores. Por otro lado, en una arquitectura p2p el cambio se notifica de manera

directa a los demás jugadores. Podemos ver como afecta un cambio en la figura 2.1



(a) Cambio por parte de un jugador en una arquitectura glscliente-servidor



(b) Cambio por parte de un jugador en una arquitectura p2p

Figura 2.1: Comparación de un cambio en un esquema genérico cliente servidor 2.1(a) y un esquema genérico p2p 2.1(b)

2.1.1. Servidor dedicado

Existen una gran cantidad de videojuegos que emplean este tipo de servidor. También se puede ver como se puede utilizar este esquema para solo algunas funcionalidades.

Como videojuegos que utilizan un servidor dedicado para mantener todo el estado del juego, notificar cambios en este y comunicar a los jugadores tenemos el World os warcraft [4], el Diablo 3 [2], el Runescape [5], el Dofus [6] o el Counter strike: global offensive [7] entre muchos otros.

Este tipo de arquitectura es utilizada para mantener una máxima consistencia del estado del juego y por eso, la utilizan la mayoría de videojuegos multijugador masivo online.

Tiene la ventaja de que es más sencillo implementar un buen sistema de seguridad. Sin embargo, no es una garantía de seguridad, ya que si se programa mal puede ser inseguro. Esto se ve al comparar el world of warcraft [4], con una seguridad prácticamente perfecta con el runescape [5], famoso por la enorme cantidad de hackers que tiene.

También se puede ver el uso de servidores dedicados para llevar a cabo tareas como autenticarse o buscar una partida. En el caso del Call of duty [8] se utiliza un servidor dedicado para permitir a los jugadores buscar una partida y un esquema de *listen server* para administrar y sincronizar esa partida entre los distintos jugadores.

2.1.2. Listen server

Se trata de un servidor alojado en el juego de un cliente. Este servidor permite a un equipo funcionar como un servidor al mismo tiempo que se juega.

Como hemos mencionado anteriormente el Call of duty [8] utiliza este esquema. Esto permite que los propios jugadores administren el juego y así se ahorren recursos. Sin embargo, es sabido que la seguridad en este tipo de esquemas no es absoluta y es posible hacer trampas.

Otra aplicación que tienen este tipo de arquitectura es para LAN parties. El poco retardo permite tener una gran experiencia. Uno de los primeros juegos en implementarlo fue el Counter Strike [9]

2.1.3. Peer to Peer

Sistema en que todos los jugadores comparten entre ellos el estado local del juego que tienen en su máquina.

Un caso muy claro en que este modelo es utilizado es en los juegos de peleas online. La latencia debe ser mínima y solo participan dos jugadores. Como ejemplo tenemos el Mortal Kombat 1 [10]. Aunque la seguridad en estos videojuegos es menor ya que todos los nodos comparten su propia información y se tiene que confiar en que no hagan trampas, se puede aprovechar el menor coste de estos en los juegos de consola, ya que por sus características son más difíciles de modificar; un ejemplo muy claro de esto es splatoon [11]

2.2. Protocolo

En el ámbito de los videojuegos online se utilizan dos protocolos: TCP y UDP.

En los videojuegos que no sean de acción y que vayan por turnos se suele utilizar el protocolo TCP. Esto se puede ver si jugamos una partida al ajedrez online [12] y se capturan los paquetes con

wireshark [13] (2.2).

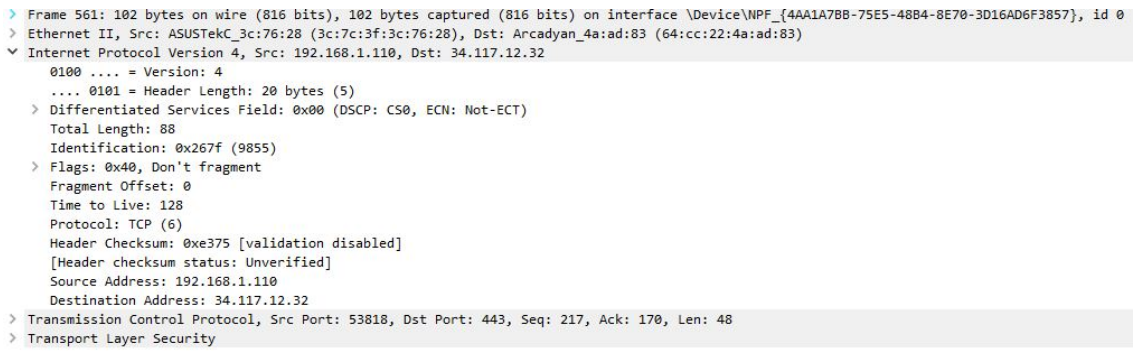
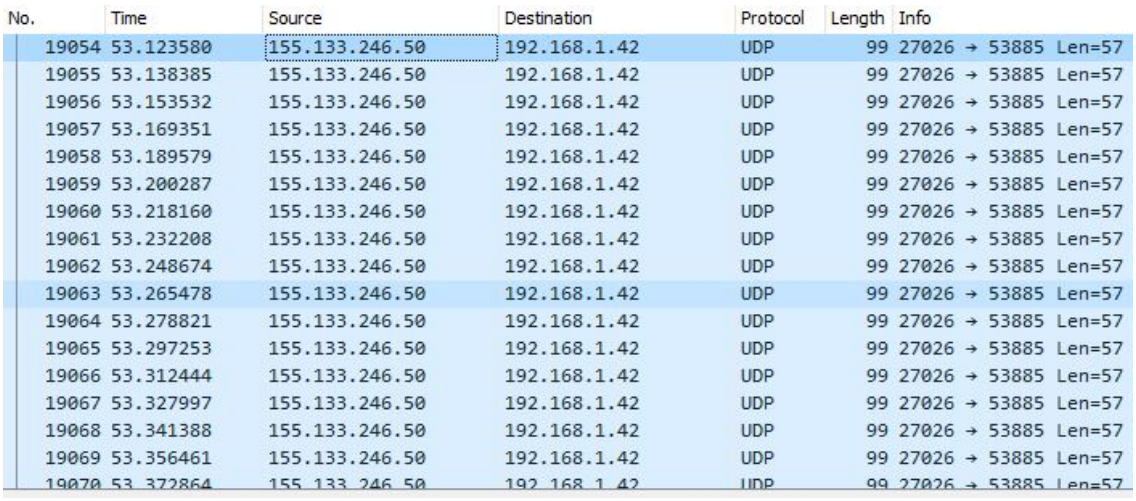


Figura 2.2: Paquetes recibidos durante una partida de ajedrez en chess.com. Se puede ver que utiliza TCP

Los videojuegos que utilizan este tipo de sistema son juegos de estrategia por turnos. Donde un fallo en la sincronización sería muy relevante. Estos tampoco requieren una actualización inmediata. Tenemos como ejemplo el ajedrez online [12], el juego de Magic: the gatheting [14] o el hearthstone [15]

Por otro lado, se sabe que el principal protocolo utilizado en videojuegos multijugador online (Y aplicaciones de tiempo real en general) es UDP. Esto es debido a que los mensajes enviados por UDP no tienen la latencia provocada por la congestión en linea o el establecimiento de la conexión entre el cliente y el servidor.

Un ejemplo de juego que usa este protocolo es el Counter Strike: Global Offensive [7]. Lo cual se puede ver muy facilmente analizando los paquetes que recibimos en mitad de una partida con Wireshark [13] (2.3).



No.	Time	Source	Destination	Protocol	Length	Info
19054	53.123580	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19055	53.138385	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19056	53.153532	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19057	53.169351	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19058	53.189579	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19059	53.200287	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19060	53.218160	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19061	53.232208	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19062	53.248674	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19063	53.265478	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19064	53.278821	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19065	53.297253	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19066	53.312444	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19067	53.327997	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19068	53.341388	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19069	53.356461	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57
19070	53.372864	155.133.246.50	192.168.1.42	UDP	99	27026 → 53885 Len=57

Figura 2.3: Paquetes recibidos durante una partida de Counter Strike: Global Offensive. Se ve que utiliza UDP

También es utilizado este protocolo en Quake [16], Diablo 3 [2]... Como podemos ver, es un están-

dar en los juegos de acción.

Finalmente, es necesario destacar que aunque un videojuego utilice el protocolo UDP para manejar el estado de juego, se utiliza TCP para acciones importantes como autenticarse, spawnear enemigos... En definitiva, paquetes que no deben perderse.

2.3. Seguridad

La seguridad en los videojuegos multijugador online es un factor de vital importancia. Si un jugador hace trampa en un juego para un solo jugador solo afectaría a su experiencia, pero en un juego online puede perjudicar desde decenas a millones de jugadores.

Explicado de forma muy resumida, las principales vulnerabilidades que tiene un videojuego online son:

- Mediante ataques de denegación del servicio. Un ejemplo son los ataques DDoS.
- Alterando el estado del juego de forma ilegal. Por ejemplo, diciendo que se tienen mejores estadísticas que las reales.
- Utilizando programas que supongan una ayuda significativa. Por ejemplo, usando aimbots.
- Obteniendo información personal o comprometedora debido a un fallo del servidor. Por ejemplo, mediante inyección sql.

A continuación detallaremos cuales son los métodos utilizados para evitar estas vulnerabilidades.

2.3.1. Ataques de denegación de servicio

Un ataque de denegación de servicio se basa en hacer inaccesible un determinado recurso o servicio. En el caso de los videojuegos online se suelen basar en atacar a los servidores o a usuarios en específico.

Evitar un ataque de denegación de servicio a un usuario es tan sencillo como ocultar en todo momento la IP de los jugadores. Esto se puede hacer de dos formas:

- Utilizando un sistema cliente-servidor que oculte esta información. Como ejemplos tenemos overwatch [17], world of warcraft [4], diablo 3 [2] entre muchos otros
- Haciendo que los jugadores utilicen una VPN.

Es realmente complicado hacer que todos los usuarios utilicen una VPN. Por esto mismo la opción utilizada suele ser ocultar a los demás usuarios esta información. Aunque se suele recomendar el uso de una VPN

Por otro lado, hacer que el servidor sea invulnerable a un ataque de denegación de servicio es complicado ya que siempre puede recibir una cantidad de peticiones que excedan su capacidad. Un

claro ejemplo de esto son los problemas para conectarse a la salida de una expansión en World of Warcraft [4] debido a la cantidad de conexiones simultáneas de los usuarios. Muchas empresas coinciden en que lo mejor es tener un buen protocolo para recuperarse de estos ataques. [18]

2.3.2. Modificaciones ilegales del estado del juego

Hemos visto que muchos juegos optan por un sistema cliente-servidor debido a la facilidad de evitar este tipo de trampas, aunque en juegos de consola se pueden utilizar sistemas p2p con buenos resultados.

Para evitar modificaciones ilegales en el caso de sistemas cliente-servidor se suelen emplear dos opciones:

- El uso de un **servidor autoritario** de forma que el cliente solo notifica los inputs y este calcula el estado del juego. De esta manera no se puede alterar el estado de manera ilegal desde el cliente. Esta es la forma utilizada por la mayoría de juegos multijugador online.
- Otra alternativa menos conocida sería el utilizar **mirror servers**. Al haber varios servidores con el estado del juego se hace más difícil engañarlos a todos. Este tipo de estructura puede dar problemas si los servidores no están correctamente sincronizados. Un ejemplo de este tipo de sistema es el quake [16].

En el caso de los sistemas p2p se utilizan sistemas de sincronización asíncrona o se hace que el juego se desarrolle en turnos y el estado sea comprobado por cada peer. Se basa en que no puedes confiar en un solo jugador, pero sí en varios de ellos. Un curioso ejemplo de juego que utiliza p2p con sincronización entre sus nodos es splatoon [11].

2.3.3. Programas externos

Para esta tarea existen diversos programas como punkbuster [19] o VAC (Valve Anti-cheat System) [20]. Se basan en controlar los procesos de escritura en memoria, por lo que tenerlos controlados puede ser una buena opción que dificulte en gran medida el uso de estos programas [21].

2.3.4. Protección de información sensible

La forma de prevenir esto es igual para todo tipo de aplicaciones en internet. No tiene características relevantes en el ámbito de los videojuegos.

La práctica que se suele utilizar de manera característica en este medio es impedir que el usuario envíe mensajes en los que figure su contraseña y recordar que no se debe facilitar información sensible. World of warcraft hace esto [4]

3

Definición

3.1. Alcance

Este proyecto pretende ofrecer un sistema que permita a los usuarios autenticarse, registrarse y jugar al videojuego de ejemplo. Para esto dispondrá de un videojuego, un cliente y un servidor.

El videojuego a utilizar es una extensión del videojuego presentado en la asignatura de introducción a la programación de videojuegos y gráficos. Este añade la posibilidad de registrarse como usuario y autenticarse. Para esto se comunica con un servidor que accede a una base de datos en mysql con la información de los usuarios (encriptada cuando así se requiera). Este videojuego instancia al cliente encargado de comunicarse con el servidor.

La comunicación con el servidor a la hora de registrarse y autenticarse se lleva a cabo empleando el protocolo de comunicación TCP.

En el momento en que se establece una comunicación TCP entre el cliente y el servidor se comparte la clave pública RSA del servidor. Esta clave es empleada para cifrar el paquete que contenga el usuario y la contraseña del jugador.

El servidor se comunica con el cliente instanciado en el videojuego. Este servidor tiene la obligación de autenticar y registrar a los distintos usuarios en la base de datos. Para esto lleva a cabo las medidas adecuadas para evitar SQL injection. Del mismo modo, tiene la tarea de crear una partida entre varios usuarios y administrarla.

Una vez comience una partida instanciada por el servidor, este se encarga de informar de los cambios que produzcan los jugadores y de llevar a cabo distintos cambios (crear enemigos, instanciar objetos...).

Durante la partida el intercambio de información entre cliente y servidor de realiza empleando el protocolo de comunicación UDP.

El servidor será autoritario y almacena las IDs e IPs de los jugadores cuando se autentiquen. Gracias a esto evita que una persona que no esté jugando envíe paquetes haciéndose pasar por un

jugador. Si la IP no coincide con el ID del usuario, el paquete es ignorado.

El juego es utilizado para mostrar el correcto funcionamiento del servidor y del cliente. Por esto mismo no tiene persistencia y el personaje del jugador se reinicia en cada partida de una manera similar a league of legends [22].

El videojuego ha sido desarrollado con unity y el servidor con el lenguaje c.

La base del servidor se basa en el tutorial creado por Tom Weiland [23].

3.2. Requisitos funcionales

3.2.1. Requisitos del servidor

- **RF-1.– Procesar petición de login**
  - El servidor debe poder procesar peticiones de login. Para esto descifrará el mensaje de login, encriptado con su clave pública, con su clave privada. Tras esto, comprobará el usuario y la contraseña introducidos con el usuario y la contraseña de la base de datos.
- **RF-2.– Enviar respuesta al login**
  - El servidor informará cuando la autenticación ha sido correcta para que el cliente pueda actualizarse.
- **RF-3.– Procesar petición de registrarse**
  - El servidor debe poder procesar peticiones de registro. Para esto introducirá en la base de datos el usuario, el correo electrónico y la contraseña cifrada después de añadir un «salt» fijo y aleatorio.
- **RF-4.– Enviar respuesta al registrarse**
  - El servidor informará si se ha procesado correctamente la petición de registrarse o no.
- **RF-5.– Simular una partida**
  - El servidor simulará una partida con todos los jugadores para asegurar la consistencia entre clientes.
- **RF-6.– Administrar el estado del juego**
  - El servidor almacenará el estado del juego (enemigos, objetos, personajes...)
- **RF-7.– Realizar cambios en el estado del juego**
  - El servidor realizará cambios en el estado del juego (instanciar enemigos, instanciar objetos...) y los notificará a los jugadores.
- **RF-8.– Procesar los cambios realizados por los jugadores**
  - El servidor actualizará el estado del juego en función a las acciones que realicen los jugadores (Eliminar un personaje, eliminar un objeto, actualizar la posición de un enemigo, actualizar la posición de un personaje...). Los jugadores solo podrán notificar de sus inputs y el servidor actualizará el estado del juego.
- **RF-9.– Informar del estado del juego**
  - El servidor notificará de las interacciones de los jugadores a los distintos jugadores.

- **RF-10.– Generar clave pública y clave privada**
  - El servidor generará una clave pública y una clave privada al iniciarse.
- **RF-11.– Ofrecer su clave pública**
  - El servidor incluirá en un paquete su clave pública cuando se inicie la comunicación con un cliente.
- **RF-12.– Acceso a la base de datos mediante sentencias preparadas**
  - El servidor solo realizará peticiones SQL a la base de datos mediante sentencias preparadas.
- **RF-13.– Comprobación de los caracteres en una lista blanca**
  - En aquellos paquetes que contengan información relativa a la base de datos se realizará una comprobación de esa información utilizando una lista blanca.

3.2.2. Requisitos de la base de datos

- **RF-14.– Almacenar la información de los usuarios**
  - La base de datos almacenará la información de los usuarios (nickname, correo electrónico y contraseña)

3.2.3. Requisitos del cliente

- **RF-15.– Solicitar clave pública**
  - El cliente le solicitará al servidor su clave pública al inicio de la comunicación.
- **RF-16.– Realizar petición de login**
  - El cliente realizará la petición de login al servidor y procesará su respuesta. Esta petición cifrará la información con la clave pública del servidor.
- **RF-17.– Realizar petición de registrarse**
  - El cliente realizará la petición de registrarse al servidor y procesará su respuesta. Esta petición cifrará la información con la clave pública del servidor.
- **RF-18.– Mostrar respuestas a las peticiones erróneas**
  - El cliente mostrará en pantalla un mensaje de error en el caso de no poder registrarse o autenticarse.
- **RF-19.– Notificar de las acciones del jugador**
  - El cliente enviará las acciones del jugador al servidor para que este pueda actualizar el estado del juego.
- **RF-20.– Procesar actualizaciones por parte del servidor**
  - El cliente actualizará la vista del juego de acuerdo a los paquetes enviados por el servidor.
- **RF-21.– Evitar los procesos de simulación de teclado o de ratón**
  - El cliente funcionará de tal manera que el uso de procesos de simulación de teclado y ratón no serán relevantes para la experiencia de otros jugadores.

3.2.4. Requisitos del videojuego

- **RF-22.– Ofrecer la GUI del cliente**
  - El videojuego implementará el cliente y ofrecerá una GUI para que el usuario pueda introducir la información requerida para las peticiones al servidor.
- **RF-23.– Responder a la interacción del usuario**
  - El videojuego reaccionará a los cambios realizados por el usuario.
- **RF-24.– Ofrecer una pantalla completamente jugable**
  - El videojuego tendrá una pantalla completamente jugable por varios jugadores.
- **RF-25.– Enemigos controlados por IA**
  - El videojuego tendrá enemigos controlados por inteligencia artificial. Estarán sincronizados entre todos los jugadores.
- **RF-26.– Objetos consumibles**
  - El videojuego tendrá objetos consumibles que podrán ser utilizados por el usuario. Se recogerán al pasar por encima de ellos.
- **RF-27.– Objetos pasivos**
  - El videojuego tendrá objetos pasivos. Se recogerán al pasar por encima de ellos.
- **RF-28.– Un personaje con dos ataques**
  - El jugador será un personaje que dispondrá de dos ataques. Este tendrá forma de esqueleto y será diferenciable de los otros jugadores.

3.2.5. Requisitos de persistencia

- **RF-29.– Estado del juego persistente**
  - El estado del juego estará almacenado en el servidor y este será el mismo para todos los jugadores que haya en la partida.
- **RF-30.– Generación de enemigos**
  - La generación de enemigos será administrada por el servidor.
- **RF-31.– Cooldown persistente**
  - El cooldown de los jugadores estará administrado por el servidor.
- **RF-32.– Manejo de desconexiones**
  - El servidor manejará las desconexiones de los jugadores para evitar que se vea algún jugador que ya está desconectado.

3.3. Requisitos no funcionales

3.3.1. Requisitos de seguridad

- **RNF-1.– Evitar inyección SQL**
  - El servidor debe evitar la inyección SQL ya que hacerlo en el cliente es inseguro. Para esto antes de realizar ninguna petición se comprobarán todos los caracteres de esta en una lista blanca y se utilizarán sentencias preparadas.
- **RNF-2.– Almacenar las contraseñas de forma segura**
  - Para aumentar las seguridad, las contraseñas se almacenarán con un salt y encriptadas. Para comprobar si la contraseña introducida es correcta se encriptará y se comprobará con el valor almacenado en la base de datos.
- **RNF-3.– Evitar trampas por parte del usuario de modo local**
  - Se dispondrá de al menos una herramienta para evitar que el jugador pueda realizar acciones ilegales (Cambiar estadísticas o el estado del juego a voluntad).
- **RNF-4.– Comprobación de los mensajes de los jugadores**
  - Se comprobará que los mensajes que informan de las acciones de los jugadores han sido generados por esos jugadores. Para ello se almacenará la IP de los usuarios y se comprobará cada paquete.
- **RNF-5.– Ocultación de los jugadores**
  - Los jugadores no se intercambiarán paquetes entre ellos.
- **RNF-6.– Descartar paquetes sospechosos**
  - El servidor descartará paquetes que no estén construidos de forma correcta. Así se evitan los paquetes creados de forma manual.
- **RNF-7.– Ocultación de las IPs de los usuarios**
  - Las IPs de los usuarios serán invisibles frente a otros usuarios.

3.3.2. Requisitos de rendimiento

- **RNF-8.– Evitar retardo en el procesamiento de los paquetes**
  - El servidor debe procesar la información de los jugadores en menos de 100ms para garantizar una buena experiencia de juego.

3.3.3. Requisitos de sistemas

- **RNF-9.– Sistemas operativos compatibles**
  - El servidor, el cliente y el videojuego serán compatibles con windows.

4

Diseño

4.1. Tecnologías y estándares

Para el desarrollo de este TFG se han utilizado diversas tecnologías y estándares. Todo el código ha sido escrito en c# ya que es el lenguaje más adecuado para usar con unity. Las consultas a la base de datos han sido realizadas con SQL y utilizando una base de datos en mysql.

Para asegurar la persistencia y el control de versiones se ha utilizado github y unity collab.

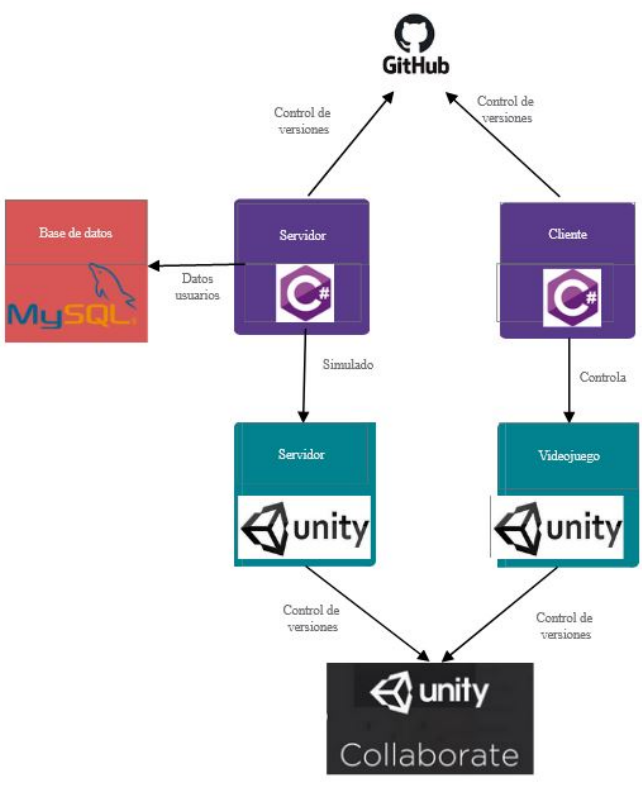


Figura 4.1: tecnologías utilizadas y relación entre ellas

4.1.1. Tutorial de Tom Weiland

La base del servidor ha sido tomada del tutorial realizado por Tom Weiland en su canal de youtube [23]. En este tutorial se implementan métodos para enviar, construir y leer los paquetes/datagramas.

Este servidor posee una clase **Packet** que contiene los métodos y los campos para administrar los paquetes. La clase **ServerSend** se encarga de construir y enviar los paquetes destinados al cliente mientras que la clase **ServerHandle** contiene lo necesario para procesar los paquetes enviados por el cliente.

La clase **Server** almacena la información de los usuarios, los métodos de conexión TCP y UDP y un diccionario con los callbacks de la clase **ClientHandle**.

Para abstraer a los usuarios tenemos la clase **Client** que almacena su información de red y un **Player** que almacena su información en el juego.

Finalmente tenemos **ThreadManager** y **NetworkManager** que controlan el hilo que corre el servidor y todo lo relacionado con la red.

De creación propia tenemos la clase **RSA** que tiene un método para obtener la clave pública como un XML y otro para descifrar. Utiliza la librería de c# **System.Security.Cryptography**.

Finalmente está la clase **DatabaseManager** que administra todo lo relacionado con el acceso y modificación a la base de datos en mysql. Cada consulta es realizada después de comprobar que todos los caracteres se encuentran en una lista blanca y mediante una sentencia preparada para garantizar una correcta seguridad.

El cliente posee una estructura similar al servidor sin el acceso a la base de datos.

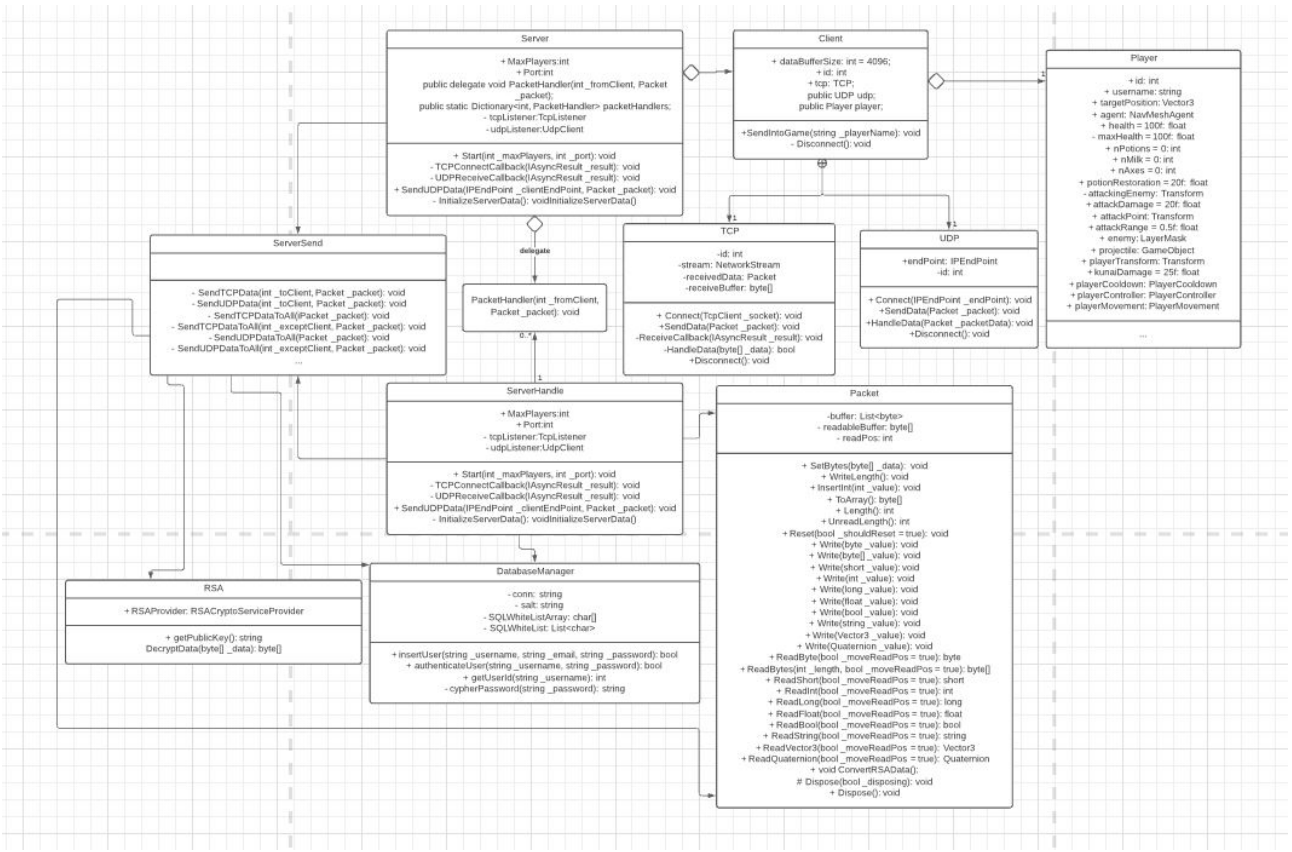


Figura 4.2: Diagrama de clases que muestra la interacción de las distintas clases del servidor

4.1.2. Unity

El cliente y el servidor corren bajo Unity. Esto permite que el servidor tenga una simulación con los datos del juego corriendo con mucho menos detalle para tener un correcto control de los jugadores y los enemigos.

El servidor corre una versión simplificada del cliente. Esto permite que no tenga que renderizar texturas o administrar sistemas de partículas. Lo que permite procesar la información más rápido.

4.1.3. Videojuego

El videojuego es una ampliación del videojuego presentado en la asignatura de *Introducción a la programación de videojuegos y gráficos*.

4.2. Manejo de la información del jugador en el servidor

Al utilizar un servidor autoritario no se permite que el jugador informe de su posición o estadísticas. Para esto solo se permiten los inputs del usuario y el servidor los administra.

4.2.1. Movimiento

El movimiento del usuario se realiza mediante clicks en el suelo al igual que Diablo 3 [2]. El servidor administra el estado de este movimiento para tener consciencia de la posición de todos los jugadores. También notifica al resto de los jugadores de los clicks realizados.

- **El jugador que realiza el click**
  1.– Obtiene la posición donde se ha hecho el click.
  2.– Envia por UDP al servidor la posición donde ha hecho click.
  3.– Actualiza en el personaje la posición donde se ha hecho click. Como consecuencia el personaje comienza a desplazarse
- **El servidor**
  1.– Recibe un paquete UDP que informa de movimiento.
  2.– Comprueba que el ID del paquete coincide con la IP del jugador. Si no coincide lo ignora.
  3.– Actualiza en su simulación la posición objetivo del jugador y esta se desplaza.
  4.– Envia un datagrama con la nueva posición objetivo de ese jugador a todos los jugadores conectados.
- **Los jugadores**
  1.– Recibe un paquete UDP que informa del movimiento de un jugador. Si el su propio movimiento lo ignora.
  2.– Actualiza la posición objetivo de ese jugador. Su personaje comienza a moverse

Este protocolo para el movimiento permite que el jugador no perciba ningún retardo entre su click y el movimiento. Los jugadores y el servidor tienen el pathfinding suministrado por un nav mesh agent (componente de unity).

Como los parámetros de posición y velocidad se encuentran en el servidor no van a haber discrepancias. Si el jugador modifica su posición o velocidad esto no se verá plasmado en el servidor y no afectará a ningún otro jugador, simplemente habrá una vista errónea debido a ese intento de hacer trampa. A pesar de que el jugador que hace trampa vea una cosa, el resto de jugadores verán una vista correcta.

4.2.2. Inteligencia artificial: Enemigos

La inteligencia artificial de los enemigos es administrada por el servidor.

Tenemos dos tipos de enemigos. Cada IA se administrará de forma distinta:

- Enemigo a melee.
- Enemigo a distancia.

Enemigo a melee

El enemigo a melee tiene tres estado:

1.– Patrolling: En el caso de que no haya ningún jugador en rango cambia de posición buscándolo.

2.– Chase Player: Cuando hay un jugador en rango se acerca a él.

3.– Attack Player: Ataca al jugador cuando está suficientemente cerca. Si está esperando el cooldown se queda quieto.

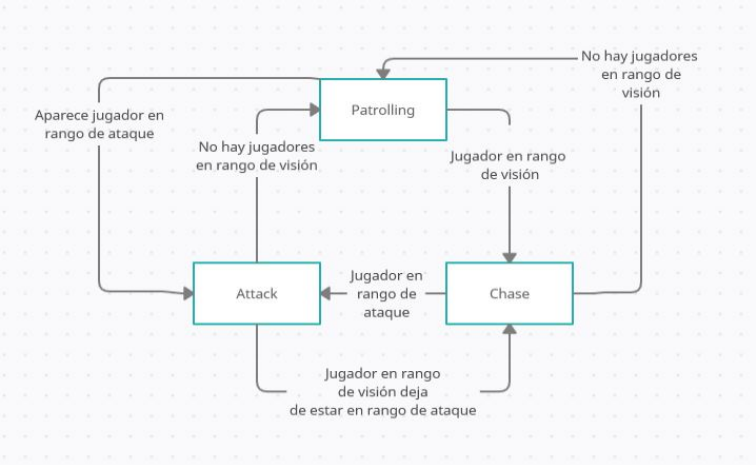


Figura 4.3: Máquina de estados que representa la IA del enemigo a melee

La IA es administrada en el servidor. Este informa de los estados a los clientes. Mantiene el jugador objetivo hasta que este se aleja demasiado:

- **Patrolling:** Jugador objetivo a null.
  - Si no tiene jugador en el rango:
    1.– Obtiene una posición objetivo nueva aleatoria para el agent.
    2.– Notifica la nueva posición objetivo a los clientes.
  - Si hay uno o más jugadores en rango:
    1.– Asigna un jugador aleatorio como el objetivo.
    2.– Notifica a los clientes el jugador objetivo.
    3.– Cambia a Chase o a Attack en función de la distancia.
- **Chase:** Hay jugador objetivo en rango de búsqueda, pero no en rango de ataque.
  - El objetivo es el jugador y lo persigue.
  - Cambia de estado si sale del rango de búsqueda o entra en el rango de ataque.
- **Attack:** Se ha acercado lo suficiente al jugador objetivo.

- Si tiene cooldown espera en estado idle.
- Cambia de estado si sale del rango de búsqueda o entra en el rango de ataque.
- Si no tiene cooldown ataca al jugador.
- Notifica el daño realizado al jugador a todos los clientes.
- Cambia de estado si sale del rango de búsqueda o entra en el rango de ataque.

Las animaciones y el comportamiento se administran en los clientes junto al servidor. Si un cliente es modificado para que un jugador no reciba daño o los enemigos no le ataquen dará igual, ya que lo importante es la simulación del servidor y la vista actualizada con esa información de los demás jugadores.

El daño recibido y hecho también es administrado por el servidor.

En el caso de los clientes se muestra el siguiente comportamiento:

- **Patrolling:** Jugador objetivo a null.
  - Animación de Patrolling activada.
  - Cambiará de posición cuando lo notifique el servidor.
  - Tendrá jugador objetivo cuando lo notifique el servidor.
- **ChasePlayer:** Hay jugador objetivo en rango de búsqueda, pero no en rango de ataque.
  - Animación de Patrolling desactivada. Running activada.
  - Incrementa la velocidad (igual que el servidor).
  - El destino es el jugador objetivo.
  - Dejará de tener jugador objetivo cuando el servidor lo notifique.
- **Attack:** Se ha acercado lo suficiente al jugador objetivo.
  - Animación idle.
  - Rotación orientada al jugador.
  - Si recibe la orden del servidor mostrará la animación de ataque.
  - La vida del jugador se actualizará en otro paquete del servidor.

Enemigo a distancia

El enemigo a distancia tiene tres estado:

1.– Patrolling: En el caso de que no haya ningún jugador en rango cambia de posición buscándolo.

2.– Attack: Cuando hay un jugador en rango le puede atacar.

3.– Get Distance: Cuando el jugador se acerca lo suficiente huye de él.

La IA es administrada en el servidor. Este informa de los estados a los clientes. Mantiene el jugador objetivo hasta que este se aleja demasiado:

- **Patrolling:** Jugador objetivo a null.
  - Si no tiene jugador en el rango:

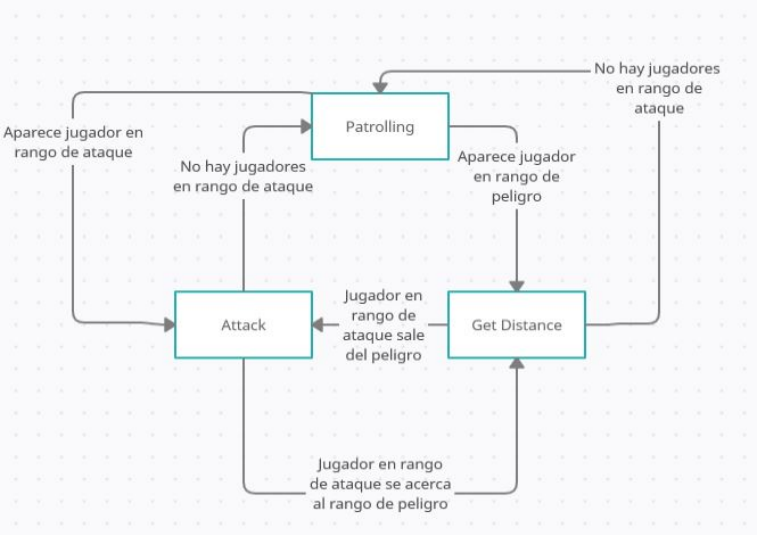


Figura 4.4: Máquina de estados que representa la IA del enemigo a melee

1.– Obtiene una posición objetivo nueva aleatoria para el agente.

2.– Notifica la nueva posición objetivo a los clientes.

- Si hay uno o más jugadores en rango:

1.– Asigna un jugador aleatorio como el objetivo.

2.– Notifica a los clientes el jugador objetivo.

3.– Cambia a Attack o a Get Distance en función de la distancia.

- **Attack:** Hay jugador objetivo en rango de ataque, pero no en el rango de peligro.
  - Mantiene el objetivo.
  - Realiza un ataque si no tiene cooldown.
  - informa del ataque realizado para que lo muestre el cliente.
  - Si le acierta, notifica el daño.
- **Get Distance:** El jugador objetivo está en el rango de peligro.
  - Huye en dirección contraria al jugador.
  - informa al cliente de la nueva posición objetivo.

Al igual que con el enemigo a melee todo aquello que puede causar problemas es administrado por el servidor para asegurar un correcto funcionamiento.

En el caso de los clientes se muestra el siguiente comportamiento:

- **Patrolling:** Jugador objetivo a null.
  - Animación de Patrolling activada.
  - Cambia de posición cuando lo notifica el servidor.
  - Tiene jugador objetivo cuando lo notifica el servidor.
- **Attack:** Hay jugador objetivo en rango de ataque, pero no en el rango de peligro.
  - Animación de idle activada.

- Espera el ataque por parte del servidor. Una vez recibido.
  1.– Realiza la animación e instancia la flecha..
  2.– El daño realizado (en el caso de impacto) lo notifica otro paquete.
- Deja de tener jugador objetivo cuando el servidor lo notifica.

- **Get Distance:** El jugador objetivo está en el rango de peligro.
  - Cuando recibe el paquete indicando la nueva posición objetivo.
    1.– Animación de correr.
    2.– Cambia la rotación para que sea en la dirección contraria al jugador.
    3.– Lo mueve hasta que llega a la dirección.

Es necesario destacar que la comprobación de si un jugador está en el rango de peligro solo es verdadera si ese jugador es el jugador objetivo.

4.2.3. Ataque cuerpo a cuerpo

El ataque cuerpo a cuerpo se puede realizar presionando a un enemigo o presionando al suelo.

En el caso de que presione al suelo:

- **El jugador que realiza el click**
  1.– Realiza la animación del ataque.
  2.– Envia un datagrama al servidor informando de que ha realizado un ataque. Este paquete informa de la rotación del personaje (importante ya que la simulación del videojuego no la tiene en cuenta).
- **El servidor**
  1.– Recibe el paquete que indica que se ha atacado al suelo.
  2.– Envia un paquete para que los demás jugadores vean la animación de ataque a melee.
  3.– Cambia la rotación del jugador que ha atacado.
  4.– Comprueba los enemigos que están en el rango de ataque del jugador.
  5.– Le resta vida a los enemigos.
  6.– Notifica de la vida restante de los enemigos afectados.
- **Los jugadores**
  1.– Reciben un datagrama que informa de un jugador y la acción de atacar. La muestran.
  2.– Cambian la vida del enemigo. Si ha muerto, ejecuta la animación de muerte.

En el caso de que presione a un enemigo solo varía el jugador que hace click:

- **El jugador que realiza el click**
  1.– Tiene como objetivo el enemigo. Se acerca a él. Cuando está suficientemente cerca le ataca.
- **El servidor**
  1.– Simula el movimiento de acercamiento. Cuando se ha acercado lo suficiente realiza lo mismo que cuando se ataca al aire en el caso anterior.

Este protocolo para el combate cuerpo a cuerpo sigue permitiendo el pathfinding y no permite que los jugadores hagan trampas diciendo que están más cerca de los enemigos de lo que están en realidad o diciendo que hacen más daño del real. Todo esto lo comprueba el servidor.

El daño realizado depende del parámetro de daño a melee del jugador. Este es almacenado en el servidor.

4.2.4. Ataque a distancia

El ataque a distancia tiene la peculiaridad de que no es simulado en el servidor debido a los recursos que consume. Hacer eso podría suponer que otras acciones tengan un retardo superior al deseado.

- **El jugador que realiza el click**
    1.– Instancia un proyectil y le aplica una determinada fuerza.
    2.– Si no impacta con un enemigo no pasa nada.
    3.– Si impacta con un enemigo desaparece. En ningún caso le hace daño al enemigo.
- **El servidor**
    1.– Recibe datagrama UDP que informa de un ataque a distancia. Este paquete contiene la rotación del jugador.
    2.– Comprueba el cooldown del jugador.
    3.– Simula el ataque a distancia:
        3.1.– Rota al jugador tal y como ha informado el paquete.
        3.2.– Instancia un raycast y comprueba si le ha dado a un enemigo.
        3.3.– Si le ha dado a un enemigo le hace el daño base del ataque a distancia.
        3.4.– Notifica a los demás jugadores si le ha hecho daño a un jugador.
    4.– Notifica a los demás jugadores del ataque a distancia. El paquete también contiene la rotación notificada.
- **Los jugadores**
    1.– Reciben un datagrama que informa de un ataque a distancia y de la ip del usuario que lo ha hecho.
    2.– Rotan a ese jugador e instancian un proyectil. El comportamiento del proyectil es el mismo en el caso del jugador que hace el click.

Este protocolo permite calcular los ataque a distancia de forma rápida y con una precisión muy similar a la que se muestra en el juego. El raycast es lanzado con un retardo para simular la distancia y el tiempo que tarda el proyectil en recorrer la distancia. El tiempo de retraso es calculado utilizando las fórmulas de un Movimiento Rectilíneo Uniforme sin tener en cuenta la fricción.

El daño que recibe el enemigo es en función del parámetro de daño a distancia almacenado en el servidor.

4.2.5. Curación

La curación se realiza pulsando la tecla p. Permite que el jugador recupere puntos de salud si tiene pociones disponibles. Para esto:

- **El jugador:**
  1.– Pulsa la tecla p.
  2.– El cliente comprueba si tiene pociones para no enviar un paquete innecesario.
  3.– Si tiene pociones disponibles envía el paquete al servidor.
- **El servidor:**
  1.– Recibe un datagrama de curación.
  2.– Comprueba si el usuario tiene o no pociones.
  3.– Si tiene pociones disponibles le añade salud.
  4.– Envía un datagrama con los nuevos puntos de salud al jugador.
  5.– Envía un datagrama indicando que ha consumido una poción.
- **El jugador:**
  1.– Recibe un datagrama con sus puntos de salud nuevos y los actualiza. También actualiza la barra de salud.
  2.– Recibe un datagrama indicando que ha consumido una poción. La elimina del contador.

La doble comprobación de la disponibilidad de pociones es para:

1.– No enviar datagramas inútiles (cliente).
2.– Comprobar que el paquete no ha sido enviado de forma ilegal (servidor) si falla la comprobación anterior.

4.2.6. Objetos

Los objetos son administrados en su totalidad por el servidor. Cuando un enemigo muere puede dejar una poción de salud o una mejora de estadísticas, protección en el caso del enemigo a distancia y daño en el caso del enemigo a melee.

Cuando un enemigo suelta un objeto:

- **El servidor:**
  1.– Comprueba mediante la generación de un número aleatorio si el enemigo suelta objeto.
  2.– Si suelta objeto lo spawnea en el servidor.
  3.– Informa a todos los jugadores de la posición y el tipo de objeto para que se muestre en el cliente.
- **Los jugadores:**
  1.– Reciben el datagrama informando del objeto.
  2.– Lo spawnean para que los jugadores lo vean.

En el caso de que un jugador pase por encima del objeto:

- **El servidor:**
  1.– Cuando un jugador para por encima de un objeto lo añade al inventario del jugador (manejado por el servidor).
  2.– Envía un datagrama para que el objeto desaparezca en los clientes.
  3.– Si es una poción le envía un datagrama al jugador que lo ha recogido para que actualice su contador de pociones.
- **Los jugadores:**
  1.– Eliminan el objeto de la escena.
  2.– El jugador que haya recogido una poción sumará 1 a su contador de pociones.

Como podemos ver todo lo relativo a los objetos se administra en el servidor. Así este puede comprobar si el usuario que intenta curarse tiene o no pociones. Del mismo modo, la mejora de estadísticas se guarda y calcula en el servidor. Así no se puede mentir diciendo que se tiene más protección o más daño que el real.

5

Desarrollo

Durante el proceso se siguió un desarrollo iterativo e incremental. En un primer momento se definieron objetivos y posteriormente se definieron las iteraciones.

Las iteraciones definidas fueron:

1.– **Base del servidor:** Consistente en establecer la base de envío de paquetes y cifrado de estos. También se desarrolló el menú inicial y el poder registrarse y autenticarse.

2.– **Spawneo de personajes:** Una vez autenticado se debía mostrar el juego y el personaje del jugador. También se pasó el servidor a unity para la simulación.

3.– **Movimiento de los personajes:** Una vez establecida la simulación del servidor se implementó el movimiento de los personajes.

4.– **Combate cuerpo a cuerpo:** Con el movimiento establecido se pasó al combate cuerpo a cuerpo. También se desarrolló el enemigo a melee.

5.– **Combate a distancia:** El siguiente paso fue desarrollar el combate a distancia. Al igual que con el combate a melee se aprovechó esta iteración para implementar el enemigo a distancia.

6.– **Objetos y desconexión de los jugadores:** Finalmente se implementó el manejo de los objetos, la curación del jugador y la desconexión de los distintos jugadores.

Todas las iteraciones que trabajaban sobre la jugabilidad fueron llevadas a cabo con la misma metodología. Esto permitió controlar los fallos individuales primero y ampliarlo a varios jugadores de manera controlada:

1.– Envío de los paquetes del jugador.

2.– Manejo de la información en el servidor.

3.– Envío de los paquetes a un solo jugador.

4.– Envío de los paquetes a varios jugadores.

5.1. Iteraciones

5.1.1. Base del servidor

Lo primero de todo fue programar la base del servidor en un entorno a parte. Para esto utilizamos Visual Studio 2019. El cliente comenzó a ser desarrollado en Unity.

Durante toda esta primera etapa se siguió el tutorial de Tom Weiland [23]. Esto permitió no perder tiempo en tareas como programar desde cero los paquetes o los sockets, que son tareas más tediosas y ya han sido realizadas en las prácticas.

A parte de la base se programaron dos módulos:

- DatabaseManager (Lado del servidor)
- RSA (Lado del cliente y lado del servidor)

DatabaseManager permitía realizar consultas de forma segura a la base de datos, ya que comprueba todos los caracteres en una lista blanca y las consultas son realizadas mediante sentencias preparadas.

RSA es un módulo que implementa cifrado RSA. En el caso del servidor permite descifrar mensajes y obtener la clave pública del servidor. En el lado del cliente permite almacenar la clave pública del servidor y cifrar los mensajes.

Con todo esto, finalmente se implementó el registrarse y autenticarse.

5.1.2. Spawneo de personajes

Con la base del servidor realizada se pasó a permitir el spawneo de personajes y a cambiar el servidor a Unity.

Lo primero fue pasar el servidor a Unity. Para esto se creó un nuevo proyecto y copiamos del videojuego:

- El terreno.
- Las tumbas, necesarias para el path finding.
- Los prefabs del jugador y los enemigos.
- Los scripts necesarios fueron añadidos cuando se necesitaban.

Una vez estaba lo necesario para la simulación se creó el objeto NetworkManager para almacenar los scripts de red. Tras esto, se crearon los paquetes para instanciar un jugador en el cliente. Este paquete se envía mediante TCP, ya que no se debe perder.

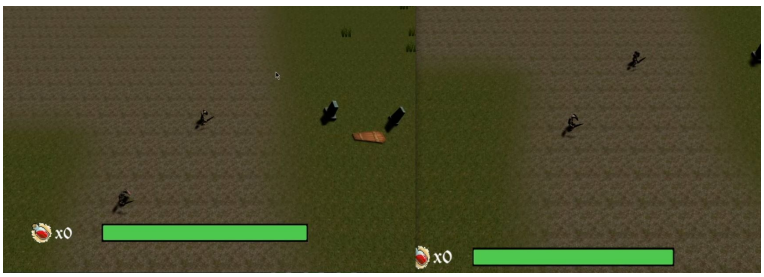
5.1.3. Movimiento de los personajes

Una vez se podían instanciar los personajes, se comenzó a programar su movimiento.

Durante el desarrollo de esta iteración se intentó actualizar la posición del personaje de acuerdo a la simulación del servidor, pero esto suponía un mayor retardo y no permitía tan buena experiencia de juego. Otra opción que se probó fue actualizar la posición del personaje que controlamos cuando el servidor confirme el paquete, pero no da tan buena sensación al jugar. Finalmente se optó por el protocolo explicado en la fase de diseño.

Esta fase concluyó con las pruebas del tercer diseño, por lo que se realizaron todas las fases de la iteración tres veces. Podemos ver el resultado en la figura 5.1.

(a) Movimiento: Antes del click



(b) Movimiento: Despues del click

Figura 5.1: Ejemplo de movimiento. Podemos ver el estado del juego en dos jugadores antes de hacer click 5.1(a) y despues de hacer click 5.1(b)

5.1.4. Combate cuerpo a cuerpo

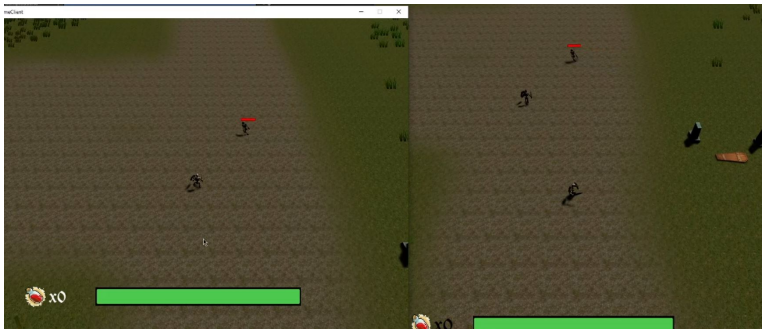
Una vez el se podía mover a los personajes a voluntad se desarrolló el combate.

No se contempló el permitir en ningún momento que los jugadores hicieran daño al enemigo en el cliente. Una opción desarrollada, probada y descartada fue el utilizar una esfera de colisión como la del jugador en los enemigos, pero produce más retardo y no se sincroniza de forma correcta. La opción desarrollada fue la detallada en la fase de diseño.

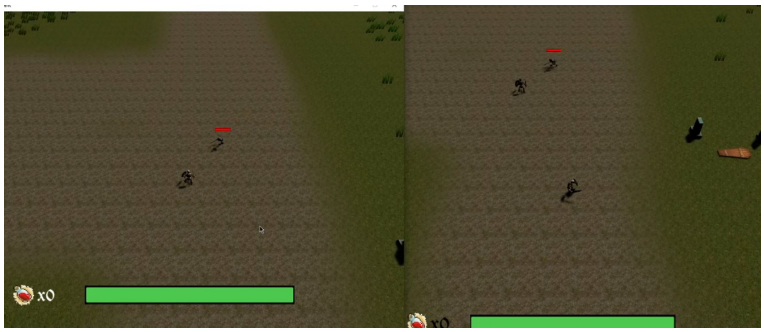
Se desarrolló el combate a melee al mismo tiempo que el enemigo a melee ya que funcionan de una manera muy parecida.

Esta fase concluyó con las pruebas del segundo diseño, por lo que se realizaron todas las fases de la iteración dos veces. Podemos ver un ejemplo de la IA en la figura 5.2 y del combate a melee en la

figura 5.3



(a) Enemigo a melee: Estado de patrulla



(b) Enemigo a melee: Estado de perseguir al jugador

Figura 5.2: Ejemplo de IA. Podemos ver al enemigo a melee buscando a un jugador 5.1(a) y persiguiéndolo cuando está en rango 5.1(b)

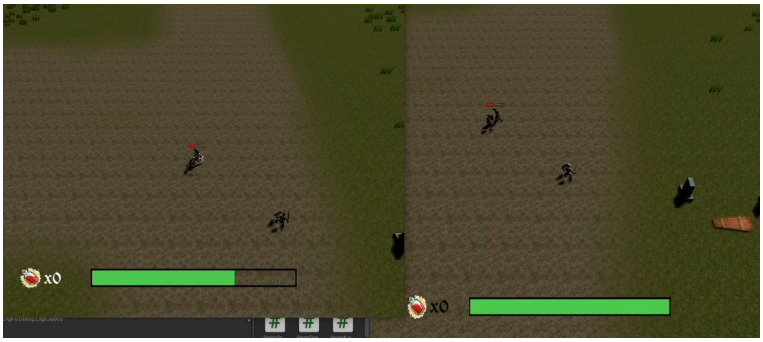


Figura 5.3: Podemos ver una imagen del desarrollo del combate a melee

5.1.5. Combate a distancia

Con el combate a melee desarrollado se procedió a desarrollar el combate a distancia.

La primera opción que se probó fue simular los proyectiles de igual manera que funcionan en el cliente, pero en la fase de pruebas vimos que un box collider trigger en los proyectiles no siempre funcionaban, aún cuando la detección de colisiones era continua. Para solucionar esto se utilizó un

Ray para simular el proyectil, esperando un poco para simular el tiempo que tarda el proyectil en llegar. De esta manera siempre se detecta la colisión cuando ocurre y no hay apenas margen de error entre el cliente y el servidor.

Esta fase concluyó con las pruebas del segundo diseño, por lo que de realizaron todas las fases de la iteración dos veces. Podemos ver un ejemplo del combate a distancia en la figura 5.4

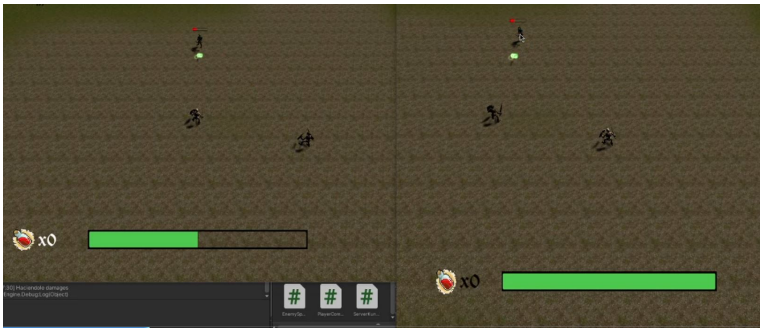


Figura 5.4: Podemos ver una imagen del desarrollo del combate a distancia

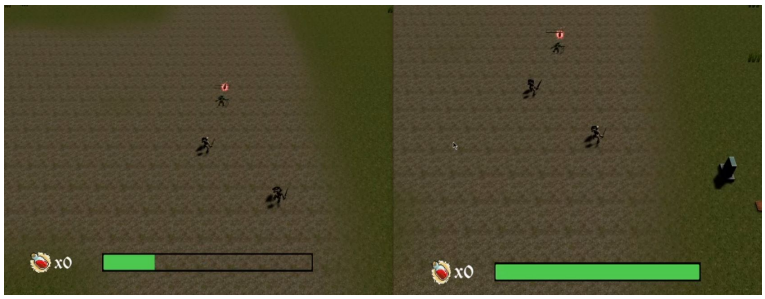
5.1.6. Objetos y desconexión de los jugadores

Con el combate a distancia implementado ya solo faltaba permitir la aparición y recogida de objetos. Así como permitir la desconexión de los jugadores.

Los objetos son spawneados por el servidor y su recogida también es administrada por el servidor. El cliente solo gestiona si se muestran o no.

La desconexión se produce al apagar el juego o morir (0 o menos puntos de salud). Al ocurrir esto se cierra la conexión y los sockets pertinentes. También se le informa a los demás jugadores de esto para que desaparezca el jugador.

Esto fue realizado en un ciclo iterativo. Podemos ver esto en la figura 5.5



(a) Objetos: Enemigo droppea un objeto para luego desaparecer

(b) Objetos: Coger objeto

(c) Objetos: Consumir poción para recuperar salud

Figura 5.5: Ejemplo de aparición de una poción al morir un enemigo 5.5(a), coger la poción y ver el contador de pociones actualizado5.5(b) y consumir esa poción para recuperar salud 5.5(c)

6

Pruebas

Durante el desarrollo de este proyecto se desarrollaron 3 tipos de pruebas:

- Pruebas sobre el correcto funcionamiento de las funciones.
- Pruebas sobre la recepción y manejo de los paquetes en la red.
- Pentesting.

6.1. Pruebas de las funciones

Las pruebas sobre el correcto funcionamiento de las funciones consistieron en pruebas caja blanca. Es decir, se probaban inputs a la función para verificar que las funciones funcionaban de forma correcta y daban error cuando tenían que darlo.

Una vez se hacían las pruebas caja blanca y se corregían los errores pertinentes siempre se probaba que estas funciones se integraban bien en el servidor y eran llamadas de forma correcta.

6.2. Pruebas en los paquetes

Para comprobar que los paquetes se enviaban y recibían de forma correcta se probó de forma individual cada tipo de paquete cuando era creado en la fase del desarrollo pertinente.

Como la base del servidor es de código libre sabemos que funciona de manera correcta, pero es muy importante cerciorarse de que los paquetes se crean cuando se tienen que crear y se manejan de forma correcta.

También se forzó la pérdida de datagramas para ver cómo puede afectar esto a la experiencia de juego. Aunque pueden producir problemas en el juego, con paquetes posteriores se suele recuperar el estado correcto del servidor.

Por otro lado, se simuló retraso de red para ver como afecta a la experiencia de juego. Se puede jugar de forma cómoda hasta unos 80ms de ping. A partir de ahí comienzan a notarse problemas a la

hora de jugar.

Esta etapa no solo sirvió para comprobar que los paquetes se envían, reciben y procesan bien. En esta etapa se pudo ver que daba más retardo y cómo se podía optimizar el funcionamiento del servidor y del videojuego para que se actualice bien.

6.3. Pentesting

Finalmente se llevó a cabo una fase de pentesting, es decir, atacar el estado del servidor como si intentáramos hacer trampas.

El atacante puede crear paquetes de forma manual y enviarlos, pero debido a como maneja el servidor la información no resultan efectivos para hacer trampas y de hecho, es hasta molesto tener que ir creándolos ya que simulan el input del jugador.

En la primera parte del pentesting fue llevada a cabo por el equipo que desarrolló el juego, por lo que no se pudo probar alterar un cliente que esté en otra IP. Sin embargo, se intentó ir a la parte que se creía más vulnerable del servidor. No se pudo vulnerar la seguridad

Para la segunda parte se le pidió ayuda al club de ciberseguridad para que ellos intentaran atacar el juego desde otros equipos sin conocer como está hecho el servidor. A continuación se explicará que hicieron.

En un primer lugar atacaron la pantalla del menú. Se intentó llevar a cabo inyección SQL y capturar los paquetes con la información de los usuarios.

La inyección SQL no resultó con éxito. Esto es debido a que se pasan todos los caracteres por una lista blanca y se utilizan sentencias preparadas. Simplemente se devolvía un error.

Los paquetes si que se pueden capturar pero están cifrados con la clave pública del servidor. Se puede obtener la clave pública de este, pero obtener la clave privada de este fue imposible. Realizar este ataque mediante fuerza bruta es increíblemente costoso, ya que la clave empleada para el cifrado RSA se calcula con números de 2048 bits.

A continuación se atacó el juego como tal. Se intentó alterar el estado de este de forma ilegal. Lo que se intentó fue:

- Alterar la posición del jugador o enemigos.
- Fingir inputs de otros jugadores.
- Modificar el inventario de los jugadores.
- Modificar las estadísticas.

Alterar la posición del jugador o enemigos fue posible dentro del cliente. Se puede cambiar donde se encuentra el jugador o los enemigos. Sin embargo, esto es algo visual que no afecta al estado del

servidor ni a otros jugadores. Podemos concluir que no se pudo realizar de forma correcta.

Se intentó enviar paquetes para simular inputs de otros jugadores. Esto sí que fue posible, pero solo cuando dos jugadores estaban jugando en el mismo equipo, ya que compartían dirección IP. No se pudo alterar a un jugador que está en otro equipo.

El inventario de los jugadores no se pudo modificar. Se pudo cambiar el contador de pociones y enviar el datagrama de curarse aunque no se tengan pociones. Sin embargo, el servidor realiza de nuevo esta comprobación, así que realmente no se pudo alterar esto.

La vida del jugador también se puede modificar de forma virtual, pero la vida real del jugador no es modificada.

La velocidad y tamaño también se puede modificar en el cliente pero de nuevo, no afecta al lado del servidor.

La conclusión obtenida es que no se pudo alterar de verdad al servidor y presenta una muy buena seguridad.

7

Resultados y trabajo futuro

Al finalizar todas las etapas del desarrollo se ha conseguido un servidor funcional, a prueba de trampas y que cumple todos los requisitos especificados. A pesar de esto, se detallarán posibles mejoras y usos del código en trabajos futuros.

7.1. Conclusiones

El servidor obtenido cumple todos los requisitos para ser un servidor efectivo y seguro.

El hecho del desarrollo de este servidor permitirá que haya un servidor de videojuegos en código abierto más, lo cual es complicado de encontrar ya que el hecho de que un código sea abierto permite que hackers y jugadores tramposos puedan encontrarle alguna vulnerabilidad. Además de expandirá en un futuro como un servidor para ser utilizado en juegos multijugador masivo online con miles de jugadores.

También se puede destacar que el hecho de utilizar un servidor sencillo como base permite que programadores nóveles puedan aprender como funciona sin una complicación excesiva.

7.2. Trabajo futuro

Como trabajo futuro se plantean las siguientes funcionalidades:

- **Uso de una base de datos no relacional:** Esto permitiría el almacenamiento de grandes cantidades de información. Así se podría mantener un sistema de inventario con una enorme cantidad de objetos.
- **Generalizar a otros motores:** Permitir un servidor que no tenga que correr en Unity y pueda ser utilizado por otros motores.
- **Protocolo ante ataques DOS:** Plantear y automatizar un protocolo de defensa ante ataques de denegación de servicio.
- **Mejora en la simulación del servidor:** Aunque la simulación de proyectiles no permite fallos de forma teórica el PING de los jugadores puede provocarlo. Se propone un sistema de simulación que tenga en cuenta el PING de los jugadores para evitar errores. Esto podría hacer que ante un mayor PING se siga teniendo una correcta

experiencia de juego.

- **Funcionamiento del servidor en varias escenas:** Ampliar el servidor para que funcione con varias escenas en el juego. Actualmente solo puede manejar una.

Bibliografía

[1] I. Luque, "Los 10 videojuegos más jugados este 2020," 2020.

[2] "Diablo 3." https://us.diablo3.com/es-mx/. Accessed: 2020-05-21.

[3] "Tom weiland." https://github.com/tom-weiland. Accessed: 2020-03-16.

[4] "World of warcraft." https://worldofwarcraft.com/es-es/. Accessed: 2020-02-17.

[5] "runescape." https://www.runescape.com/splash. Accessed: 2020-05-21.

[6] "dofus." https://www.dofus.com/es/mmorpg/descargar. Accessed: 2020-05-21.

[7] "Counter strike: Glocal offensive." https://store.steampowered.com/app/730/CounterStrike_Global_Offensive/. Accessed: 2020-04-13.

[8] "Call of duty." https://www.callofduty.com/es/ghosts. Accessed: 2020-02-17.

[9] "Counter strike." https://store.steampowered.com/app/10/CounterStrike/?l=spanish. Accessed: 2020-05-21.

[10] "Mortal kombat 11." https://www.mortalkombat.com/. Accessed: 2020-02-17.

[11] "Splatoon." https://splatoon.nintendo.com/es/. Accessed: 2020-03-16.

[12] "chess." https://chess.com/. Accessed: 2020-05-21.

[13] "wireshark." https://www.wireshark.org/. Accessed: 2020-04-13.

[14] "Magic arena." https://magic.wizards.com/es/mtgarena?utm_source=blindferret&utm_medium=cpi&utm_campaign=arn&utm_term=network&utm_content=opb-network-display-all-native-arn-opb-na01-m99. Accessed: 2020-05-21.

[15] "hearthstone." https://playhearthstone.com/es-es. Accessed: 2020-05-21.

[16] "Quake." https://quake.bethesda.net/es/. Accessed: 2020-03-16.

[17] "overwatch." https://playoverwatch.com/es-es/. Accessed: 2020-04-13.

[18] "Ddos." https://phoenixnap.com/blog/prevent-ddos-attacks. Accessed: 2020-05-21.

[19] "Punkbuster." https://www.evenbalance.com/. Accessed: 2020-03-16.

[20] "Vac." https://support.steampowered.com/kb/7849-RADZ-6869/. Accessed: 2020-03-16.

[21] "cheats." https://stackoverflow.com/questions/960499/how-to-prevent-cheating-in-our-multiplayer-games. Accessed: 2020-02-17.

[22] "League of legends." https://na.leagueoflegends.com/es-es/. Accessed: 2020-03-15.

[23] "Tom weiland tutorial." https://www.youtube.com/watch?v=uh8XaC0Y5MA&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5. Accessed: 2020-03-16.

Definiciones

aimbot Programa que realiza de manera automática la tarea de apuntar a los objetivos..

clave privada Clave conocida por un solo miembro de la comunicación. Se emplea para el cifrado asimétrico.

clave pública Clave empleada para el cifrado asimétrico. No se oculta.

cliente-servidor Tipo de red donde hay dos niveles jerárquicos: Los que solicitan servicio y los que lo proporcionan.

DDoS Ataque de denegación de servicio distribuido. Interrumpe el correcto funcionamiento del objetivo saturando su tráfico de red.

lista blanca Lista de caracteres se pueden utilizar.

listen server Servidor que corre en el mismo proceso que el cliente.

p2p Tipo de red donde los distintos equipos en ella se encuentran al mismo nivel jerárquico.

pentesting Proceso de atacar a tu propio sistema con el fin de encontrar problemas en la seguridad de este.

SQL injection Ataque que modifica, borra u obtiene información privada de una base de datos.

VPN Red privada virtual. El tráfico de red es mediado por un intermediario: El servidor VPN.

Apéndices

A

Requisitos del videojuego

En este apéndice se detallan las características del personaje que controla el jugador y las características de los enemigos que no han sido explicadas en la fase de diseño.

A.1. Jugadores

A.1.1. Manager

Para permitir una correcta experiencia tenemos un script PlayerManager encargado de controlar el estado y acciones del jugador. PlayerManager tiene la función de administrar los inputs del jugador y llamar a ClientSend cuando detecte un input; por esto mismo solo está presente en el prefab LocalPlayer.

A.1.2. Personajes

LocalPlayer

El juego está pensado para la existencia de un personaje principal controlado por el jugador mediante *clicks* y atajos de teclado.

Consiste en un esqueleto humano bípedo que porta una espada. Tiene una textura blanca.

PlayerManager se ocupará de controlar el movimiento del jugador y el sistema de combate asociado a él. Cada una de las instancias del juego tiene su propio LocalPlayer. Este implementa PlayerManager.

Player

Puede hacer lo mismo que LocalPlayer pero es controlado por otro jugador. Sus acciones vienen dadas por el servidor, que son un reflejo de las acciones de los otros jugadores.

No tiene un componente PlayerManager para evitar que estos otros agentes reaccionen a los inputs

de otros jugadores.

Tiene una textura más oscura para diferenciarlo de LocalPlayer

A.1.3. Características de los personajes

Velocidad de movimiento

La velocidad por defecto no se modifica mediante ningún objeto. Se administra en el servidor, lo que da la posibilidad de cambiarla mediante objetos en un futuro.

Vida

Los jugadores poseen una cantidad máxima de Puntos de Salud (ps) de 100.

En el caso de que el jugador llegue a **0 PS** morirá y se desconectará al jugador. Los demás jugadores verán desaparecer su personaje.

El jugador puede recoger Pociones que le permitirán recuperar PS.

El servidor administra la variable de los puntos de salud de:

1.– Recuperar PS al consumir pociones y no sobrepasar la vida máxima.

2.– Si los PS llegan a 0 o por debajo se se notifica de la muerte a todos los jugadores.

3.– Informa al jugador de los cambios en sus puntos de salud.

Daño

Al igual que con la vida el personaje tiene un daño Base (DB) al que se le podrá añadir un Daño extra.

No hay ningún efecto de aumento de daño de forma temporal.

El servidor administra las variable relativas al daño y las modifica de forma pertinente.

El daño infligido por el ataque a distancia y el ataque a melee es el mismo daño base. El daño a melee se modifica con objetos.

A.1.4. Estados del jugador

El jugador tendrá cuatro estados. **Quieto, ataque, desplazándose y muerto**. Cada estado tendrá asociada una o varias animaciones.

El paso de un estado a otro y el mostrar la correcta animación será tarea de PlayerManager en

el caso del jugador local. En el caso de otros jugadores lo notifica el servidor para tener un mayor feedback visual de las acciones de los otros jugadores.

Idle

En este estado tenemos la posibilidad de:

- **Atacar** cambiando al estado **Atacar**.
- Permanecer quieto.
- **Morir** por bajada de puntos de salud. Lo que nos llevaría al estado de **muerte**.

En este estado se muestra la animación idle.

Se pasa a este estado cuando no se esté ejecutando la acción asociada de los otros estados excepto del estado muerte que no tiene salida.

Ataque

En este estado atacamos. Podemos atacar a melee o a distancia, pero no podemos movernos mientras atacamos.

De este estado solo podemos volver a idle, pero inmediatamente podremos pasar a otros estados.

Este estado tiene dos animaciones asociadas.

- **Animación de ataque a melee.**
- **Animación de ataque a distancia.**

Desplazamiento

Estado que indica que nos desplazamos de un punto a otro.

En este estado tenemos la posibilidad de:

- Atacar cambiando al estado atacar y terminando el movimiento, por lo que al terminar el ataque pasaremos a idle.
- Terminar el movimiento y pasar a idle.
- Morir por bajada de puntos de salud. Lo que lleva al estado de muerte.

Este estado solo tiene asociada la animación de desplazamiento.

Muerte

Estado especial al que se pasa mediante la transición de muerte. Tras esto no podremos interactuar con el personaje.

Estos estados que componen el comportamiento del jugador y definen cómo serán las animaciones asociadas al personaje.

A.1.5. Objetos

Pociones

Los enemigos dropearán este ítem acumulable que se emplea para recuperar PS. La cantidad se muestra en el cliente, pero el servidor lo administra para evitar trampas.

Objetos pasivos

Se incorporan objetos pasivos que mejoran las características del jugados tales como daño y resistencia

Estos objetos no tendrán representación visual, tal solo modificarán las características del jugador.

La cantidad de objetos que tiene el jugador va administrada por el servidor. La forma en que estos afectan también se administra en el servidor.

A.2. Enemigos NPCs

Los enemigos son spawneados por el objeto vacío llamado scripter. Este objeto se encarga de hacer aparecer enemigos cada cierto tiempo hasta un máximo de enemigos en escena.

A.2.1. Enemigo a melee

De este enemigo se crea un prefab para poder spawnearlo por el mapa.

Su Inteligencia artificial viene detallada en la sección de diseño.

Su velocidad es similar a la del jugador, pero la aumenta para perseguirlo.

Su daño es relativamente bajo, ya que el jugador tiene que poder enfrentarse a varios de estos enemigos. Por cada golpe deberá quitar **15 PS** al jugador. Este valor viene administrado por el servidor en el momento en que hace daño a un jugador.

Su Velocidad de ataque será suficiente como para poder asestar 2 golpes antes de morir. El tiempo de espera entre ataques también es administrado por el servidor

Tienen una pequeña probabilidad de proporcionar un aumento de daño al jugador. Para esto dropearán un objeto que lo permita. También tienen una probabilidad de spawnear una poción. En ambos

casos está controlado por el servidor.

Tiene cuatro animaciones asociadas:

- **Idle:** Se muestra cuando espera el cooldown para atacar a un jugador y está parado.
- **Ataque:** Se muestra cuando ataca a un jugador.
- **Andar:** Se muestra en el estado de patrullar.
- **Correr:** Se muestra en el estado de perseguir a un jugador.

A.2.2. Enemigo a distancia

De este enemigo se crea un prefab para poder spawnearlo por el mapa.

Su Inteligencia artificial viene detallada en la sección de diseño.

Su velocidad será un 30 % menor que la velocidad base del jugador. Así se le puede perseguir. Esta velocidad viene dada por el servidor.

Su daño es relativamente alto, ya que es muy sencillo de esquivar. Así supone una gran amenaza si te descuidas.

Su velocidad de ataque es similar a la del enemigo a melee.

Tiene una pequeña probabilidad de proporcionar un aumento de defensa al jugador. Para esto, dropea un objeto que lo permita.

Tiene cuatro animaciones asociadas:

- **Idle:** Se muestra cuando espera el cooldown para atacar a un jugador y está parado.
- **Ataque:** Se muestra cuando ataca a un jugador.
- **Andar:** Se muestra en el estado de patrullar.
- **Correr:** Se muestra en el estado de huir de un jugador.


UAM
UNIVERSIDAD AUTONOMA
DE MADRID