Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Videojuegos en red: prototipo con técnicas y procesos desde el punto de vista del programador de un videojuego

ESCUELA POLITECNICA SUPERIOR

Autor: Pablo Largo Rubio

Tutor: Antonio Moratilla Ocaña

2023

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado
Videojuegos en red: prototipo con técnicas y procesos desde el punto de vista del programador de un videojuego

Autor: Pablo Largo Rubio

Tutor/es: Antonio Moratilla Ocaña

TRIBUNAL:

Presidente: Iván González Diego

Vocal 1º: Ángel Javier Álvarez Miguel

Vocal 2º: Antonio Moratilla Ocaña

FECHA: 6 de julio de 2023

Índice de contenido

Resumen ... 1
Palabras clave ... 1
Abstract ... 2
Keywords ... 2
Glosario de acrónimos y abreviaturas. ... 3
1. Introducción ... 4
1.1. Planteamiento del trabajo ... 4
1.2. Objetivos ... 4
2. Clasificación de los videojuegos en red ... 5
2.1. Géneros ... 5
2.2. Elementos en la parte red de los videojuegos ... 7
2.2.1. Características de red en los videojuegos ... 7
2.2.2. Capa de transporte de red ... 8
2.2.3. Capa de ejecución ... 10
2.2.4. Capa de servicios y gestión ... 10
2.3. Tipos de arquitecturas de red en los videojuegos ... 11
2.3.1. Peer-to-peer ... 12
2.3.2. Servidor dedicado ... 14
2.4. Niveles ... 15
3. Técnicas y procesos de red en los videojuegos ... 18
3.1. Técnicas de compensación de latencia ... 18
3.1.1. Técnicas de compensación de latencia basadas en feedback ... 21
3.1.2. Técnicas de compensación de latencia basadas en predicción ... 22
3.1.3. Técnicas de compensación de latencia basadas en la manipulación del tiempo 25
3.1.4. Técnicas de compensación de latencia basadas en el ajuste del mundo del juego 27
3.2. Matchmaking ... 28
3.3. Técnicas antitrampas ... 32
3.3.1. Métodos antitrampas implementados en el lado del servidor ... 34
3.3.2. Métodos antitrampas implementados en el lado del cliente ... 37
4. Diseño de los prototipos de videojuegos en red ... 40
4.1. Diseño de prototipo de shooter multijugador en tercera persona ... 41
4.2. Diseño de prototipo de juego de cartas multijugador basado en turnos ... 45
5. Implementación de los prototipos de videojuegos en red en Unity ... 47
5.1. Implementación del prototipo de videojuego shooter multijugador en tercera persona ... 47
5.1.1. Fase de formación en las herramientas a utilizar ... 47
5.1.2. Fase de desarrollo del prototipo de videojuego en red ... 50
5.2. Implementación del prototipo de videojuego de cartas multijugador basado en turnos 69
5.2.1. Fase de formación en las herramientas a utilizar ... 70
5.2.2. Fase de desarrollo del prototipo de videojuego en red ... 71
6. Conclusiones ... 88
7. Trabajo futuro ... 89
Referencias ... 90
Anexo I: Presupuesto ... 93

Índice de figuras

Fig. 1. Estructura del software de un videojuego en red [6] 7
Fig. 2. Ejemplo de un esquema de conexión de los elementos de un videojuego en red [6] 8
Fig. 3. Esquema de la arquitectura de red peer-to-peer directo [6] 12
Fig. 4. Esquema de la arquitectura de red cliente-servidor con *host* [6] 13
Fig. 5. Esquema de la arquitectura de red cliente-servidor con host y un retransmisor [6] 14
Fig. 6. Esquema de la arquitectura de red servidor dedicado [6] 15
Fig. 7. Esquema de clasificación de los videojuegos en red en base a su complejidad [5] 16
Fig. 8. Esquema de red común en los videojuegos con espacios de juego persistentes [5] 18
Fig. 9. Tiempo de respuesta y consistencia en un videojuego en red con arquitectura cliente-servidor [3] 20
Fig. 10. Árbol jerárquico de las técnicas de compensación de latencia en los videojuegos en red [3] 21
Fig. 11. Representación de la técnica de ocultación de la latencia [3] 21
Fig. 12. Escenario en el que se puede aplicar la técnica de exposición de latencia [3] .. 22
Fig. 13. Representación de la técnica de interpolación [3] 23
Fig. 14. Factores para la predicción en la técnica de extrapolación [3] 23
Fig. 15. Representación de la técnica de ejecución especulativa [3] 24
Fig. 16. Ejemplo de uso de la técnica de deformación temporal [3] 25
Fig. 17. Ejemplo de uso de la técnica de retardo de salida [3] 26
Fig. 18. Ejemplo de uso de la técnica de asistencia de control [3] 27
Fig. 19. Ejemplo de uso de la técnica de escalado de atributos [3] 28
Fig. 20. Arquitectura común en un sistema de *matchmaking* [6] 29
Fig. 21. Arquitectura alternativa de *matchmaking* propuesta por Unity [6] 29
Fig. 22. Estructura común de un paquete UDP en los videojuegos en red [15] 35
Fig. 23. Representación del funcionamiento del controlador antitrampas basado en el kernel [15] 39
Fig. 24. Ejecución de un servidor dedicado en el prototipo de shooter multijugador en tercera persona 51
Fig. 25. Elección del modo de *build* en el editor de Unity [37] 52
Fig. 26. Método para crear una sesión de juego en el prototipo de shooter multijugador en tercera persona 53
Fig. 27. Método *Start()* de la clase *NetworkRunnerHandler* del prototipo de shooter multijugador en tercera persona 53
Fig. 28. Ejemplo de fichero txt para que el servidor dedicado cree la sesión de juego en el prototipo de shooter multijugador en tercera persona 54
Fig. 29. Métodos de la interfaz *INetworkRunnerCallbacks* para ser ejecutados en el lado del servidor del prototipo shooter multijugador en tercera persona 54
Fig. 30. Menú inicial del prototipo de shooter multijugador en tercera persona 55
Fig. 31. Menú de opciones del prototipo de shooter multijugador en tercera persona ... 56
Fig. 32. Método de unión e un cliente a un *Lobby* en el prototipo de shooter multijugador en tercera persona 57
Fig. 33. Implementación del método *OnSessionListUpdated* de la interfaz *INetworkRunnerCallbacks* en el prototipo de shooter multijugador en tercera persona 57
Fig. 34. Ejemplo de error de conexión mostrado al usuario en el prototipo de shooter multijugador en tercera persona 58

Fig. 35. Implementación del método para unir a un cliente a una sesión de juego en el prototipo de shooter multijugador en tercera persona...59
Fig. 36. Diseño del *prefab* del personaje de cada usuario en el prototipo de shooter multijugador en tercera persona...59
Fig. 37. Ejemplo de estado de juego tras unirse a una sesión de juego en el prototipo de shooter multijugador en tercera persona 60
Fig. 38. Sistema de inputs del prototipo de shooter multijugador en tercera persona....61
Fig. 39. Estructura para enviar los inputs a Photon Fusion en el prototipo de shooter multijugador en tercera persona...61
Fig. 40. Método para la ejecución de acciones del personaje en base a inputs en el prototipo de shooter multijugador en tercera persona...62
Fig. 41. Método para regenerar a un personaje de un usuario en el prototipo de shooter multijugador en tercera persona...62
Fig. 42. Método con la funcionalidad de disparo del arma en el prototipo de shooter multijugador en tercera persona...63
Fig. 43. Acción de disparo en el prototipo de shooter multijugador en tercera persona 64
Fig. 44. Implementación del comportamiento de un proyectil en el prototipo de shooter multijugador en tercera persona...64
Fig. 45. Situación de impacto de un disparo en el prototipo de shooter multijugador en tercera persona...65
Fig. 46. Método para gestionar los impactos de disparos en el prototipo de shooter multijugador en tercera persona...66
Fig. 47. Método para gestionar la muerte de un personaje en el prototipo de shooter multijugador en tercera persona...67
Fig. 48. Muerte de un personaje en el prototipo de shooter multijugador en tercera persona 68
Fig. 49. Interfaz de usuario al inicio de la partida en el prototipo de shooter multijugador en tercera persona 68
Fig. 50. Configuración de la interpolación en el controlador de movimiento del personaje en el prototipo de shooter multijugador en tercera persona 69
Fig. 51. Método que implementa la creación de una sesión de juego en el prototipo de videojuego de cartas multijugador basado en turnos 72
Fig. 52. Método para implementar la autenticación el prototipo de videojuego de cartas multijugador basado en turnos...73
Fig. 53. Método para gestionar la desconexión de los clientes en el prototipo de videojuego de cartas multijugador basado en turnos 73
Fig. 54. Método que implementa la unión a una sesión de juego para los jugadores en el prototipo de videojuego de cartas multijugador basado en turnos 74
Fig. 55. Mensaje informativo al usuario mientras busca sesión en el prototipo de videojuego de cartas multijugador basado en turnos 75
Fig. 56. Método para gestionar los fallos de conexión en el prototipo de videojuego de cartas multijugador basado en turnos...76
Fig. 57. Ejemplo de tablero del usuario con información sobre una carta en el prototipo de videojuego de cartas multijugador basado en turnos...77
Fig. 58. Implementación del funcionamiento de una carta la pulsar sobre ella en el prototipo de videojuego de cartas multijugador basado en turnos 78
Fig. 59. Implementación de la funcionalidad de coger carta del “deck” en el prototipo de videojuego de cartas multijugador basado en turnos 79
Fig. 60. Implementación del método para obtener el turno inicial en el prototipo de videojuego de crtas multijugador basado en turnos 81

Fig. 61. Implementación del método que se ejecuta en los cambios de turno de los jugadores en el prototipo de videojuego de cartas multijugador basado en turnos 82
Fig. 62. Implementación del método para restablecer la parte local del cliente en el prototipo de videojuego de cartas multijugador basado en turnos 83
Fig. 63. Implementación de la gestión del ataque por el servidor en el prototipo de videojuego de cartas multijugador basado en turnos 84
Fig. 64. Implementación del método para generar la carta utilizada por el oponente en el prototipo de videojuego de cartas multijugador basado en turnos 85
Fig. 65. Situación de tablero en la que un jugador recibe un ataque en el prototipo de videojuego de cartas multijugador basado en turnos 85
Fig. 66. Situación de tablero en la que un jugador realiza una defensa en el prototipo de videojuego de cartas multijugador basado en turnos 86
Fig. 67. Implementación de la gestión de la defensa por el servidor en el prototipo de videojuego de cartas multijugador basado en turnos 87
Fig. 68. Implementación del método para la resolución de un turno en el prototipo de videojuego de cartas multijugador basado en turnos 87

Índice de tablas

TABLA I: Diferencias entre las características de Photon PUN, Bolt y Fusion 44
TABLA II: Comparación de las características de red de los dos prototipos a implementar 47
TABLA III: Coste de los materiales de software y hardware utilizados en el proyecto. 93
TABLA IV: Coste de la mano de obra en el proyecto 94
TABLA V: Presupuesto total del proyecto 94

Resumen

En el momento en el que los programadores se enfrentan al objetivo de desarrollar un videojuego en red, tienen que afrontar diversos desafíos como que arquitectura de red es la más recomendable, como evitar que la experiencia de juego de lo usuarios se vea afectada por la latencia o las trampas, como administrar las sesiones de juego…

Este trabajo analiza las distintas posibilidades que tienen los desarrolladores para implementar la parte red de un videojuego y compara la implementación de dos prototipos con géneros muy diferentes, haciendo uso de las herramientas Unity, Photon Fusion y Netcode for GameObjects.

Palabras clave

Unity, programación de videojuegos en red, latencia, antitrampas, matchmaking

Abstract

When programmers face the objective of developing an online videogame, they have to deal with several challenges such as which network architecture is the best to use, how to prevent the user's gaming experience from being affected by latency or cheats, the management of game sessions...

This project analyses the different possibilities that developers have to implement the network part of a videogame and compares the implementation of two prototypes with different genres, using Unity, Photon Fusion and Netcode for GameObjects.

Keywords

Unity, online videogames programming, lag, anti-cheat, matchmaking

Glosario de acrónimos y abreviaturas.

- RTS: Real Time Strategy (Estrategia en Tiempo Real)
- MOBA: Massive Online Battle Arena (Arena de Combate Masivo en Línea)
- RPG: Role Player Game (Juego de Rol)
- MMORPG: Massive Multiplayer Online Role-Playing Game (Juego de Rol Multijugador Masivo en Linea)
- NPC: Non-Player Character (Personaje No Jugable)
- FPS: First Person Shooter (Shooter en Primera Persona)
- TPS: Third Person Shooter (Shooter en Tercera Persona)
- IP: Internet Protocol (Protocolo de Internet)
- PC: Personal Computer (Ordenador Personal)
- TCP: Transmission Control Protocol (Protocolo de Control de Transmisión)
- UDP: User Datagram Protocol (Protocolo de Datagramas de Usuario)
- REST: Representational State Transfer (Transferencia de Estado Representacional)
- HTTTP: Hypertext Transfer Protocol (Protocolo de Transferencia de Hipertexto)
- RPC: Remote Procedure Call (Llamada a Procedimiento Remoto)
- IA: Inteligencia Artificial
- API: Application Programming Interface (Interfaz de Programación de Aplicaciones)
- CPU: Central Processing Unit (Unidad Central de Procesamiento)
- UI: User Interface (Interfaz de Usuario)

1. Introducción

La industria del videojuego hace años que se instauró como una de las más importantes dentro del sector de ocio, teniendo un volumen de negocio y personas dentro del sector que era impensable en épocas atrás, tanto a nivel de usuarios como de trabajadores.

Actualmente, gran parte de los videojuegos tienen algún tipo de soporte de red. Todo esto nos lleva a que gran parte de la población del planeta sea capaz de jugar a un videojuego en prácticamente cualquier sitio y contra cualquier persona del mundo. Para que todo esto sea posible se requieren una gran cantidad de personas encargadas de estudiar, desarrollar e implementar estos videojuegos con soporte de red, encargándose de que los usuarios puedan tener la mejor experiencia de uso posible.

Para poder implementar correctamente la parte red de un videojuego, se deben aplicar una serie de técnicas y procesos de red, lo cual depende del género de videojuego que se pretenda desarrollar. Los programadores de videojuegos en red deben tener en cuenta factores como la latencia, las trampas, el matchmaking, las distintas arquitecturas de red existentes… Con este conocimiento deben seleccionar las técnicas y procesos de red que más convengan para el videojuego que desean desarrollar.

1.1. Planteamiento del trabajo

El planteamiento de este trabajo comienza con el estado del arte, en el que se hace una investigación y estudio teórico de los tipos de juego existentes y sus características, las distintas arquitecturas de red disponibles y las técnicas y procesos de red que se aplican en la industria de los videojuegos (técnicas de compensación de latencia, técnicas antitrampas, sistemas de matchmaking…).

Tras la fase de estado del arte en el documento, se encuentra la fase de diseño, en la que se planifica la implementación de dos prototipos mediante el motor de videojuegos Unity. Se deciden conceptos como su género, su arquitectura de red, las técnicas y procesos de red que se van a implementar y las herramientas de red que se utilizarán en la implementación. En esta fase se realiza una formación en Unity y las distintas herramientas de red.

Por último, se encuentra la fase de desarrollo del proyecto, en la cual se implementan en Unity los dos prototipos de videojuego en red planificados. Tras esto, se describen las conclusiones que se han obtenido en la realización del trabajo.

1.2. Objetivos

Este trabajo de fin de grado busca plasmar algunas de las técnicas y procesos investigados previamente en dos prototipos de videojuegos en red mediante el motor Unity y un framework de comunicación en red. Se tratarán las distintas arquitecturas de red y los distintos tipos de juegos, estudiando las técnicas y procesos de red disponibles para la implementación en cada uno de ellos.

También se abarcará, como se ha comentado, la implementación de dos prototipos en los que se programe un tipo de juego con las técnicas y procesos de red más adecuados para el género elegido. El objetivo de esto de esto es llevar a cabo una comparativa de manera real entre distintos tipos de videojuegos en red que requieran de características de red distintas, aplicando los conceptos investigados y estudiados previamente a escenarios reales.

Otro de los desafíos que presenta este trabajo es el aprendizaje en el uso del motor de videojuegos Unity debido a mi completa inexperiencia en el uso de motores de videojuegos. También se llevará a cabo un estudio de los distintos frameworks de red disponibles para implementar la parte red de un videojuego en Unity. Tras el estudio, se elegirá la herramienta que se considere más conveniente para cada prototipo.

Se pretende mostrar el proceso al que se deben enfrentar los programadores de videojuegos en red para implementar la parte red de un videojuego para que este tenga la mejor experiencia de juego posible para los usuarios, debiendo tener en cuenta una gran cantidad de factores.

2. Clasificación de los videojuegos en red

En este primer apartado del documento, se van a explicar distintas formas de clasificar los videojuegos en red, basándose en distintos factores como el género al que pertenecen, la arquitectura de red de la que se componen o en base al nivel de complejidad del desarrollo de la parte red del juego.

2.1. Géneros

Una de las formas más comunes de clasificar los videojuegos es en base al género al que pertenecen, el cual se establece por características como el tipo de jugabilidad o la temática del juego. A continuación, se explican los distintos géneros que se pueden encontrar en los videojuegos en red [1] [2]:

- **Juegos de acción**
    - Lucha: juegos en los que los jugadores (suelen ser dos) se enfrentan entre sí hasta que sólo uno quede con vida. Este tipo de videojuegos requiere de un tiempo rápido de reacción y de muchos movimientos seguidos en muy poco tiempo.

- **Juegos arcade**: juegos clásicos que, en general, están caracterizados por tener unas mecánicas y unos gráficos muy simples. No se suele requerir de una gran precisión debido a la simplicidad de las mecánicas.

- **Juegos de estrategia**: este tipo de juegos implican una gran reflexión y análisis sobre los objetos, habilidades y acciones que ofrece el juego. Esto ayudará a tomar

las decisiones óptimas para vencer a los demás jugadores, siendo clave la táctica en estos videojuegos. Los principales subgéneros de este género son:

- Estrategia en tiempo real (RTS): en este subtipo de videojuegos de estrategia habrá que tomar decisiones de forma rápida y continuada, ya que se puede ser atacado o atacar a otro jugador en cualquier momento. En los combates los jugadores pueden actuar simultáneamente por lo que el tiempo de reacción es muy importante. Existen numerosos juegos que se incluyen en este subtipo, cómo en los que cada jugador tiene un poblado, el cual debe defender de los ataques de otros jugadores y, a su vez, debe atacar a otros poblados para conseguir más recursos para mejorar el suyo.

- Multiplayer Online Battle Arena (MOBA): es un subgénero de los RTS, el cual se basa en partidas en las que, generalmente, combaten dos equipos de jugadores entre sí en un mapa con el objetivo de destruir la base enemiga. Una gran coordinación y trabajo en equipo dará más probabilidades de ganar la partida.

- Juego de estrategia basado en turnos: en este subtipo, sólo un jugador actuará a la vez, teniendo bastante tiempo para decidir cuál será la siguiente jugada y sin tener mucha relevancia el tiempo de reacción. Un ejemplo es cualquier videojuego de cartas.

- **Juegos deportivos**: son videojuegos que simulan deportes, cómo partidos de fútbol, baloncesto, tenis… o carreras de coches o motos. Este tipo de videojuegos es requerido de reacciones rápidas y continuas al estar jugando contra otros jugadores, pudiendo actuar varios jugadores de manera simultánea.

- **Juegos de rol (RPG)**: en este tipo de videojuegos se avanza a través de una historia en la que el personaje del jugador va evolucionando, teniendo características propias, de las cuales dependerá el desarrollo del juego.

    - Massive Multiplayer Online Role Playing Game o MMORPG: se trata de un RPG jugado en línea que está compuesto por un mapa de grandes dimensiones lleno de muchos jugadores, cada cual tiene un personaje personalizado que va evolucionando. La actividad del resto de jugadores influirá en el personaje, ya que todos los jugadores pueden interactuar entre sí para diversas actividades.

- **Shooter**: se denomina a un videojuego como shooter a aquel en el que se ven involucradas armas y el cual consiste en combatir con esas armas contra otros jugadores en línea o personajes NPC.

    - Shooter en primera persona (FPS): este subtipo se centra en acercarse a una experiencia real, ya que la cámara del juego muestra la visión desde los ojos del personaje. Se basa en partidas en las que multitud de jugadores combaten entre sí y en las que los jugadores con más reflejos suelen salir victoriosos, por lo que las reacciones rápidas son vitales.

    - Shooter en tercera persona (TPS): este subgénero del shooter se centra en mostrar al personaje en una vista desde arriba de este, basándose también en

partidas con multitud de jugadores y en las que los reflejos de cada jugador son muy importantes.

2.2. Elementos en la parte red de los videojuegos

En esta sección del documento se van a describir las distintas características de la parte red de los videojuegos y como pueden llegar a influir en estos. Además de esto, se van a explicar las capas de red presentes en la estructura del software de un videojuego.

2.2.1. Características de red en los videojuegos

Las principales características de red en las que se tiene que centrar el desarrollo de un videojuego son las siguientes [6]:

- **Latencia**: tiempo de respuesta desde que el jugador ejecuta una acción hasta que su efecto es visible.
- **Escalado**: número de jugadores que debe alojar una partida.
- **Seguridad**: dificultad para que el juego sea hackeado, evitando que los jugadores hagan trampas.
- **Coste**: coste económico del desarrollo del juego.
- **Complejidad**: dificultad de implementación de la arquitectura de red del videojuego.
- **Alcance**: lugares a los que se desea llegar a comercializar el videojuego.

En el momento de diseñar un videojuego en red, habrá que tener todos los factores anteriores en cuenta para poder elegir la arquitectura de red (se verán los disitntos tipos en el siguiente apartado) que más convenga para la situación y la forma de desarrollar cada capa de la estructura del software. Estas decisiones influirán directamente en la clasificación del videojuego, ya que llevarán a que el videojuego esté clasificado en un tipo de arquitectura de red, un nivel de complejidad de red y un género específicos. La estructura del software de un videojuego se puede estructurar en varias capas, las cuales se muestran en la **Figura 1**.

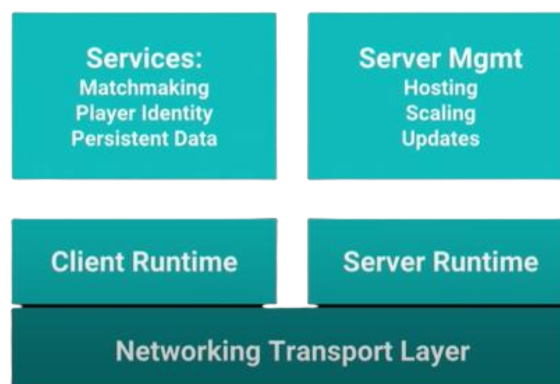


Fig. 1. Estructura del software de un videojuego en red [6]

En los siguientes subapartados, se van a explicar cada una de las capas de la estructura del software presentes en la **Figura 1**, empezando desde abajo y terminando la explicación teórica con las capas superiores.

2.2.2. Capa de transporte de red

La capa de transporte de red se encarga de la comunicación entre el hardware de un equipo e Internet a traves de bits. Para la comunicación entre los distintos dispositivos de un videojuego multijugador en red, existen varios componentes presentes en cada dispositivo:

- Dirección IP: se trata de una dirección única que identifica a cada dispositivo en la red, la cual está compuesta por 32 bits numéricos y se utiliza en el protocolo IP al poder utilizarse como un identificador del dispositivo en las comunicaciones.

- Puerto: se trata de una interfaz mediante la cual el dispositivo ofrece los servicios de transmisión de datos, ya sea para enviar o para recibirlos. Cada dispositivo dispone de 65536 puertos, los cuales pueden utilizarse para diversos servicios en la red. Para comunicarse con el resto de jugadores en el videojuego en red, utilizarán uno de ellos.
- Socket: es el elemento que funciona como punto final en la comunicación con el resto de jugadores, estando formado por una pareja compuesta por la dirección IP y el puerto que utilizará el dispositivo para la comunicación de red en el videojuego.

A continuación, se puede observar un ejemplo gráfico de comunicación en un videojuego en red mediante los elementos que se han descrito, teniendo el juego un servidor como punto central, el cual hará de nexo entre todos los dispositivos:

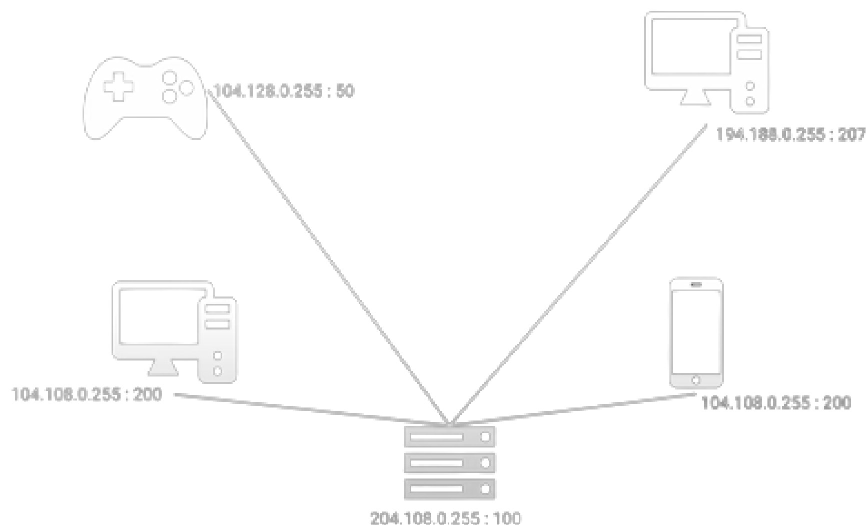


Fig. 2. Ejemplo de un esquema de conexión de los elementos de un videojuego en red [6]

Cada dispositivo (consola, PC, servidor y móvil) de la **Figura 2** tiene una única línea de conexión, la cual se estalece con el servidor, como ya se ha mencionado previamente. Además, cada dispoitivo tiene asociado una dirección IP junto al puerto con el que está estableciendo la conexión.

En el protocolo IP (utilizado por las direcciones IP como ya se ha mencionado), los dispositivos se comunican entre sí haciendo uso de paquetes/datagramas, los cuales están compuestos por los siguientes elementos:

- Cabecera: contiene la información necesaria para llevara a cabo la transmisión del paquete entre el emisor y el receptor, como por ejemplo, los dos sockets involucrados, indicando así de que dirección IP y puerto proviene el paquete y a que dirección IP y puerto va destinado.

- Área de datos: en el caso de los videojuegos multijugador en red, esta área contiene información que afectará al estado del juego, provocando cambios en este. Por ejemplo, cuando un jugador realiza una acción, esta estará descrita mediante distintos campos presentes en el área de datos.

Estos paquetes pueden ser enviados mediante dos protocolos de transimisón de datos distintos que hacen uso del protocolo IP y están presentes en la capa de transporte [7][8]:

- TCP: es un protocolo orientado a conexión, ya que su funcionamiento se basa en la conexión entre dispositivos para poder comprobar que todos los paquetes llegan de forma secuencial en orden, por lo que garantiza la entrega de datos. Cada vez que el receptor recive un paquete, se lo comunicará al emisor. Además, el protcolo TCP se encarga de detectar y corregir errores si los hay y también dispone de un control de congestión, que evita saturar al receptor mediante el uso de un búfer de recepción, el cual si está lleno, descartará el resto de paquetes entrantes. Debido a que este protocolo comprueba la recepción de los paquetes, si un paquete no llega porque el búfer de recepción o porque se ha perdido en el proceso de transferencia, este será reenviado, respetando siempre el orden de los paquetes.

- UDP: este protocolo no está orientado a conexión, ya que los dispositivos no se conectan para la transferencia de datos, el emisor simplemente envía un flujo de datos continuo sin orden y sin relizar ninguna comprobación de errores ni de recepeción. Al hacer uso del protcolo UDP, el encabezado de los paquetes es considerablemente más pequeño que al uilizar TCP (8 bytes ene vez de 20 bytes), ya que no tienen que realizar comprobaciones.

Por tanto, como principales diferencias entre los disitintos protocolos de la capa de transporte, TCP es más fiable al realizar comprobaciones de errores, recepción de paquetes y de tráfico de datos, mientras que UDP es un protocólo que proporcionará una mayor velocidad de transmisión de datos al no tener que realizar comprobaciones y tener paquetes con un encabezado más pequeño.

Esto lleva a que el protcolo TCP se utilice en aplicaciones que necesitan una alta fiabilidad en la entrega de los datos, mientras que el protocolo UDP se utiliza en servicios que priorizan una alta velocidad y eficiencia y que se pueden permitir la pérdida de paquetes, como son los videojuegos multijugador en red y la retransmisión en streaming de video y audio. En el caso del UDP, si se necesita fiabilidad en la entrega de datos, esta se podrá implementar en capas suepriores, como la capa del cliente.

2.2.3. Capa de ejecución

La siguiente capa de la estructura del software de un videojuego en red, es la parte de ejecución del cliente o servidor, en la cual se alojará todo el código adicional que estos ejecutan, pudiendo ser Unity utilizado para implementar ambas partes:

- El código de la parte de ejecución del cliente se encargará de que el juego se vea fluido y estable en la parte cliente (el dispositivo del jugador), para lo cual tendrá que compensar problemas como el retardo de la comunicación entre el cliente y el servidor o el cliente y el resto de clientes (dependiendo de la arquitectura de red del videojuego).

- El código de la parte de ejecución del servidor (o host dependiendo de la arquitectura) se encargará de la autoridad del estado del juego, es decir, comprobará que las acciones de los jugadores que recibe en los paquetes enviados por los clientes, son acciones viables teniendo en cuenta el estado del juego actual, para evitar las trampas o hacks. También se encargará de realizar y mantener todas las conexiones con los clientes. Un ejemplo en un shooter podría ser un caso en el que un cliente necesite saber si un jugador determinado está muerto, lo cual estará en la base de datos del host o servidor y será la parte de ejecución del servidor la encargada de realizar la consulta y enviar la respuesta al cliente.

2.2.4. Capa de servicios y gestión

La última capa trata de los servicios en la parte cliente (matchmaking, manejo de de la identidad del usuario y los datos persistentes…) y de la gestión del servidor (actualizaciones, ecalado, alojamiento del servidor…). Estos servicios pueden ser administrados o no administrados. Los servicios administrados (llevados a cabo por plataformas como Unity o Google Cloud) se encargan de las actualizaciones y el mantenimiento, además de ofrecer una disponibilidad 24/7. Con los servicios no administrados serán los propios creadores del videojuego en red los que tendrán que encargarse de esas tareas. Estos servicios podrán comunicarse mediante dos métodos distintos:

- REST: este método de comunicación consiste en el envío de un petición de cliete y la espera de la recepción de una respuesta del servidor a esa petición. Para ello, se utilizará HTTP, tanto sus verbos (GET, POST, PUT, DELETE…) cómo sus codigos de error estándar.

- RPC (llamdadas a procedimiento remoto): este método hace uso de un flujo de comunicación entre el cliente que hace la petición y el servidor que da la respuesta, pudiendo de esta forma enviar varias respuestas ante una sola petición o hacer varias peticiones sin tener que volver a establecer una nueva conexión. Esto hace que el RPC tenga un mayor rendiemiento que el REST con menores tiempos de resupuesta, aunque a cambio los errores se muestran en el tiempo de compilación y son personalizados, no haciendo uso del HTTP que es más sencillo.

El método de comuncación REST puede implementarse en una capa superior al RPC, por lo que no necesariamente hay que elegir el uso de uno de ellos. REST está enfocado a un uso más general al usar el estándar HTTP, mientras que RPC es más personalizable y es muy buena opción para utilizarlo en videojuegos, debido también a su gran rendimiento

y la cantidad de opciones que ofrece en su implementación, aunque hay que tener en cuenta que no tiene un estándar. Por ejemplo, cuando un jugador solicita unirse a una partida, su dispositivo enviará una solicitud y el host le puede enviar varias respuestas cada X tiempo para informar del estado de la búsqueda de partida (esto se conoce como matchmaking y se explicará detalladamente más adelante).

Se va a proceder a explicar en un mayor detalle el funcionamiento de la tecnología RPC, paso a paso [9]:

- Antes de nada hay que conocer lo que son los stubs, las cuales son instancias presentes tanto en el cliente como en el servidor, que se enacrgarán de hacer parecer al programador que la ejecución de los métodos siempre se hacen a nivel local en el cliente, aunque ese código se este ejecutando realmente en el servidor. Esto se consigue gracias a que cuando el programdor necesita invocar un método que no está presente en el cliente, lo hará como con cualquier otro metodo, poniendo los paramétros y el nombre de la función.

- Después el stub del cliente se encargará de crear un mensaje con esos argumentos y realizará un proceso conocido cómo *marshalling*, el cual consiste en serializar los datos estrucuturados a un formato secuencial establecido para que cumpla con el protocolo RPC. Una vez terminado, la interfaz de red del dispositivo del cliente lo enviará por TCP o UDP al servidor.

- Una vez llega el mensaje al servidor, este se manda al stub del servidor, el cual realiza el proceso de *unmarshalling,* mediante el cual se desempaquetan los parámetros envíados por el cliente.

- A continuación, el stub del servidor transfiere los parámetros al servidor y este se encargará de ejecutar la función en cuestión a nivel local con los parámetros recibidos.

- Tras la ejecución, el resultado se envía al stub del servidor, el cual se encargará de repetir el proceso en dirección contraria, es decir realizará el proceso de marshalling con el resultado y se lo transmitirá a la interfaz de red del servidor para que se lo envie a la del cliente.

- Tras llegar al cliente, el stub realiza el proceso de *unmarshalling* y procesa el resultado recibido, envíandolo a la parte del código del cliente que había invocado el procedimiento. Tras recibir el resultado, el código del cliente continurá la ejecución, teniendo por tanto, el mismo comportamiento que si hubiese invocado a un procedimiento presente a nivel local en el cliente.

2.3. Tipos de arquitecturas de red en los videojuegos

Una vez se ha explicado todos los componentes en los que se basa el desarrollo de un videojuego en red y las capas en las que está dividida su parte red, se va a proceder a descibir como se pueden clasificar los videojuegos basándose en los distintos tipos de arquitecturas de red que se pueden encontrar, las cuales dependen principalmente de los

elementos presentes (clientes, servidores, hosts…) y de la forma en que estos se comunican [6].

2.3.1. Peer-to-peer

Esta arquitectura de red está compuesta por clientes y no contiene un servidor dedicado (pero si hay variantes con un host que hace la función de servidor y cliente a la vez). Sin embargo, se disponen de tres variantes en esta arquitectura:

- Peer-to-peer directo: es la variante clásica, en la cual los clientes se comunican directamente entre sí y estan todos conectados con todos. La implementación de esta variante supone una gran dificultad debido a que toda la lógica del juego se encuentra en el cliente, el cual recibirá mensajes de multitud de otros clientes (el resto de jugadores) al mismo tiempo, los cuales tendrá que interpretar, llevando esto a inconsistencias y tiempo de respuesta elevados. Además, la complejidad de que el mundo del videojuego esté sincronizado en todos los clientes es muy alta.

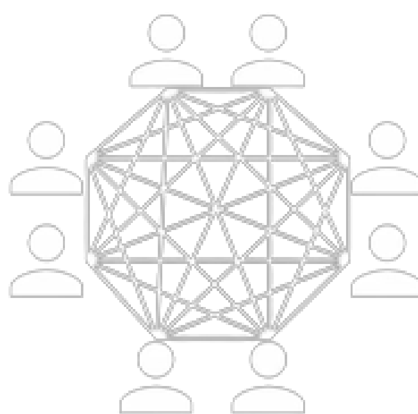


Fig. 3. Esquema de la arquitectura de red peer-to-peer directo [6]

En la **Figura 3**, se puede observar que cada cliente tiene N-1 líneas de conexión, siendo N el número de clientes/jugadores conectados a la sesión. Es decir, tendrá una línea de conexión con cada jugador presente en la partida. Además, en este tipo de arquitectura, no existe la capa de software de gestión del servidor, ya que no existe ni un host ni un servidor. Debido a todas las complejidades e inconsistencias que tiene esta arquitectura, prácticamente no se usa en la actualidad en los videojuegos en red.

- Cliente-servidor con *host*: la segunda variante se basa en la elección de uno de los clientes como el *host* de la partida. Esto lleva a que todos los jugadores se conecten a ese *host* y se evita que todos los jugadores tengan que estar conectados con todos. Sin embargo también tiene sus deventajas, ya que el jugador que sea el *host* será el que tenga que procesar los mensajes de todos los clientes y el que tenga que enviar las respuestas a todos. Además, el *host* tendrá ventaja respecto al resto de jugadores al no tener latencia (tiempo de respuesta de red). Con esta variante también puede suceder que el jugador que es el *host* decida abandonar la partida, en cuyo caso esta se acabará o se pausará, teniendo en el segundo caso que realizar una migración de *host* (la cual supone una gran complejidad de implementación), en la que la partida permanecerá pausada hasta que todos los jugadores se hayan

reconectado al nuevo jugador que actuará como *host*. Esto decrementa notablemente la calidad de la experiencia de juego al interrrumpir la partida por completo.

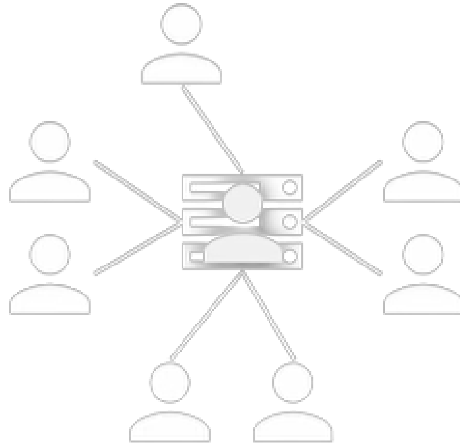


Fig. 4. Esquema de la arquitectura de red cliente-servidor con *host* [6]

En la **Figura 4** se puede observar que en este tipo de arquitectura de red los clientes solo necesitan una línea de conexión, la cual se lleva a cabo con el host de la partida. El host, en cambio, necesita tener N-1 líneas de conexión, como ocurre con cada cliente en la arquitectura peer-to-peer directo.

- Cliente-servidor con *host* y un retransmisor: la última variante de arquitectura de red *peer-to-peer* se basa también en un cliente que actua cómo *host*, aunque incluyendo también un retransmisor de confianza. Este retransmirsor actuará como intermediario entre el *host* y el resto de jugadores, lo cual evitará la mala experiencia con los firewalls de los dispositivos que puede suponer la conexión directa entre clientes, además de que el *host* ya no verá las direcciones IP de todos los jugadores, por lo que estas no estarán expuestas. Además, gracias a que el retransmisor tiene un mayor alcance, se podrá llegar a jugadores de distancias mayores, ya que en las dos variantes anteriores los jugadores de las partidas se encuentran con limitaciones en la distancia que puede haber entre ellos (además del gran incemento en la latencia). La principal desventaja de esta variante es que esto supone una latencia extra, al tener que pasar los datos siempre por un dispositivo de más.

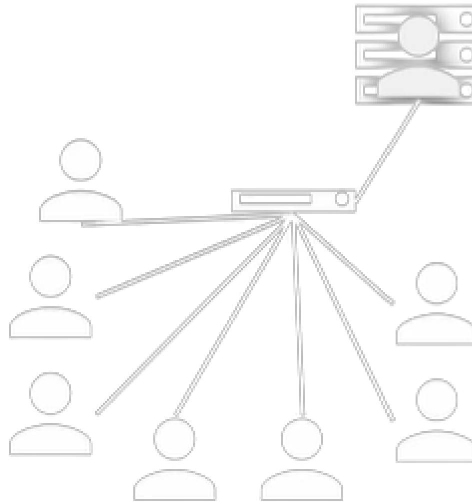


Fig. 5. Esquema de la arquitectura de red cliente-servidor con host y un retransmisor [6]

En la **Figura 5** se puede contemplar que, a diferencia de en la arquitectura de red cliente-servidor con *host*, el host ya no necesita tener N-1 líneas de conexión, si no que solo necesitará tener una línea, al igual que el resto de los clientes, que tendrán una línea con el retransmisor también, lo cual aligera notablemente la carga de proceso en el dispositivo del *host.*

La principal ventaja de la arquitectura de red *peer-to-peer* y sus variantes es el coste, ya que no se necesitan infraestructuras extras a los dispositivos de los jugadores (a excepción del retransmisor en la última variante). Sin embargo, son muchas las desventajas de este tipo de arquitectura en los videojuegos en red, cómo las limitaciones en el alcance y en el escalado (número de jugadores en una partida) debido a que se depende totalmente de los dispositivos de los clientes.

2.3.2. Servidor dedicado

Esta arquitectura de red tiene un servidor controlado por los propietarios del videojuego en red, el cual reducirá considerablemente la carga de los clientes al disponer de su propio código y sus propios servicios. Aportará diversas ventajas frente a la arquitectura *peer-to-peer*:

- Una latecia más consistente e igualitaria entre todos los jugadores y una conexíon a internet mucho más rápida del servidor en comparación con el *host.*
- La escalabilidad es mucho mayor, ya que no dependerá de los dispositivos de los jugadores, si no de cómo de potente sea el servidor y de cuantos servidores se disponga, lo cual es controlable al depender de los propietarios del videojuego en red.
- Un mayor alcance, al poder llegar a todos los jugadores del mundo, teniendo la posibilidad de añadir diversos servidores para ellos.
- Evita la exposición de las direcciones IP y aporta una gran seguridad al juego, ya que el servidor podrá controlar que los jugadores no hacen trampas al comprobar si las acciones que hacen son sospechosas de el uso de *hacks* o no.

A pesar de ser una arquitectura de red más consistente y que mejora notablemente la experiencia de juego de los usuarios, habrá que tener en cuenta que lleva un mayor coste para los desarrolladores del videojuego en red debido al coste de la infraestructura extra que suponen los servidores. También habrá que tener en cuenta la complejidad extra de implementación de la parte red, al tener que manejar los servidores y sus servicios.

La arquitectura de red con un servidor dedicado es la más uilizada en los videojuegos, debido a que es la que mejores prestaciones puede dar (consistencia, seguridad, latencia, escalabilidad, alcance…) y que puede ser mejorada por los desarrolladores del videojuego, ya que la infraestructura de los servidores depende de ellos mismos.

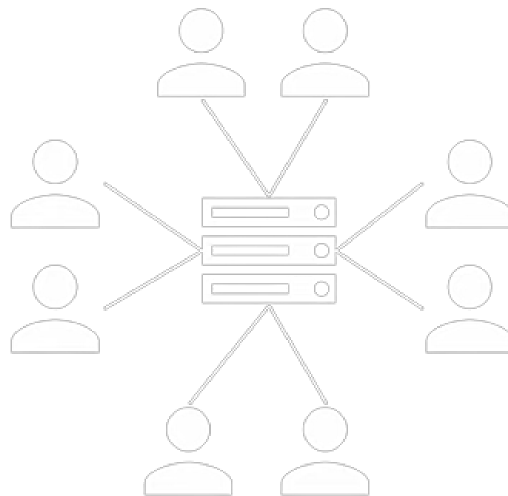


Fig. 6. Esquema de la arquitectura de red servidor dedicado [6]

En la **Figura 6** se puede ver un esquema en el que se minimiza el número de líneas de conexión a una por cliente con el servidor dedicado, sin ningún retransmisor de por medio y sin nigún cliente que tenga que ejercer la función de *host*.

2.4. Niveles

La última forma de clasificar los videojuegos en red se basa en el nivel de dificultad que supone el desarrollo de la parte red, teniendo en cuenta factores como la frecuencia de interacción de los jugadores, la cantidad de elementos en las partidas o la latencia mínima necesaria para que el videojuego sea estable.

Los videojuegos en red se pueden clasificar en 4 niveles establecidos en la conferencia Unite Berlin 2018 [5] y los cuales se pueden observar en la **Figura 7**. En esta figura se pueden observar 4 niveles, ordenados de arriba abajo en base al nivel de complejidad de la parte red del videojuego, siendo el nivel 4 (“L4” en al imagen) el más complejo. Cada nivel incluye las características de red del nivel anterior, es decir, los videojuegos pertenecientes al nivel 4 contendrán también las características de red del nivel 1, 2 y 3 a parte de las suyas propias, lo que le hará ser el nivel más complejo.

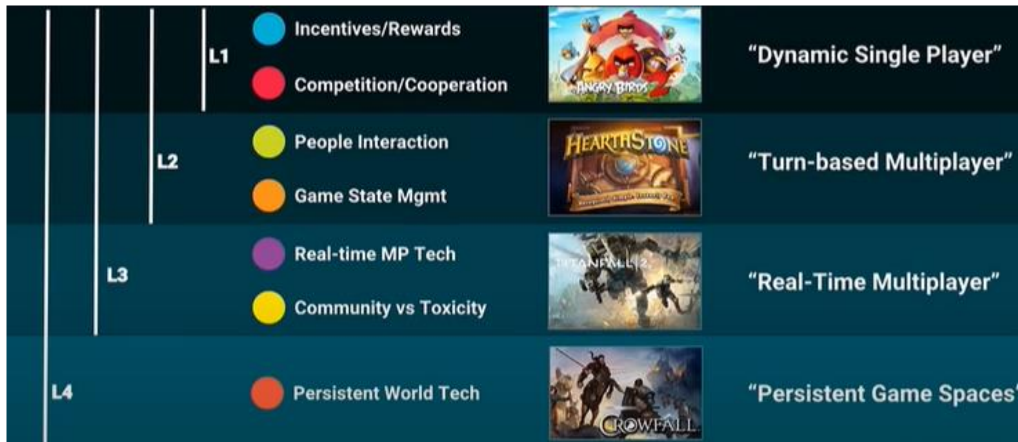


Fig. 7. Esquema de clasificación de los videojuegos en red en base a su complejidad [5]

El número de interacciones entre los usuarios y el tipo de sincronización necesaria en el estado del juego de los distintos jugadores son factores claves para determinar la complejidad de red, ya que a mayor nivel pertenezca un videojuego, mayor número de interacciones y mayor exactitud en la sincronización entre jugadores se necesitará.

A cotinuación, se detallan las características de las que se compone cada nivel de complejidad de red:

- **Videojuegos dinámicos para un jugador** (nivel 1): los juegos pertenecientes a este nivel tienen una mecánica en la que los jugadores realizan acciones o juegan partidas de forma individual pero esas acciones tendrán repercusión para el resto de los jugadores. Esa repercusión puede verse reflejada en tablas de puntuación o en premios que sólo consigan los jugadores que realicen mejores puntuaciones/actuaciones en sus partidas individuales, introduciendo de esta forma la competitividad con el resto de los usuarios en el juego.

  También puede haber cooperación entre los distintos usuarios, la cual no se verá de forma simultánea. Un ejemplo de cooperación en este nivel de juegos sería un combate contra un personaje del juego (que puede ser un personaje creado por el juego o un personaje que pertenece a otro jugador), en el que ese personaje tiene un número de vida específico y esa vida puede ser reducida más rápido cuántos más jugadores ataquen al personaje. Cada jugador no verá al resto de jugadores, si no que el ataque que cada uno proporcione se sumará a los ataques del resto (aunque uno haya atacado antes que el otro o incluso aunque uno de ellos ya no esté atacando) sin haber ninguna interacción directa entre los jugadores.

- **Videojuegos multijugador basados en turnos** (nivel 2): los juegos incluidos en este nivel se basan en partidas multijugador asíncronas, es decir, en ningún momento dos jugadores actuarán a la vez. Cada jugador sólo podrá actuar en su turno, no pudiendo haber dos jugadores con su turno activo a la vez. En estos juegos, por tanto, ya existe la interacción directa entre los jugadores, ya que los jugadores se mostrarán dentro de una misma partida, pudiendo incluso comunicarse a través de un chat en el juego.

  Antes de que tengan efecto las acciones llevadas a cabo en un turno, estas serán comprobadas por un sistema antitrampas (se explican más adelante en el

documento). En los videojuegos pertenecientes a este nivel no se requieren de tiempos de respuesta rápidos, ya que cada jugador tiene un gran tiempo para tomar decisiones sin depender de lo que estén haciendo otros jugadores en ese mismo momento.

- **Videojuegos multijugador en tiempo real** (nivel 3): los juegos pertenecientes a este nivel están basados en partidas en las que todos los jugadores interactúan con todos de forma simultánea, teniendo que introducir, por tanto, tecnologías capaces de abordar las partidas multijugador en tiempo real sincronizando el estado de juego de todos los jugadores. Las tecnologías y la arquitectura de red implementada deben de ser capaz de dotar al videojuego de consistencia en las partidas, seguridad contra las trampas/hacks, escalabilidad respecto al número de jugadores y un tiempo de respuesta bajo, ya que las reacciones y reflejos serán claves para que los jugadores puedan hacer sus mejores partidas.

  Cada partida es independiente y tiene una duración determinada. Para abordar estos problemas, se suele utilizar una arquitectura de red de servidor dedicado, ya que la que se utilizaba antiguamente ("peer to peer") no es capaz de dotar a los videojuegos en red en tiempo real de las características mencionadas previamente.

  En este nivel también se introduce el concepto de comunidad, el cual se basa en la interacción de grupos de jugadores, ya sea por chat en las propias partidas o dentro del juego o incluso en foros ubicados fuera del juego. Se debe intentar controlar la comunidad del juego, pudiendo llevar a cabo esto mediante bloqueos a los usuarios que interactúen de una forma indebida y dando la oportunidad a la comunidad de crear sus propios grupos, para que los jugadores puedan interactuar y jugar partidas con otros usuarios con los que se sientan cómodos, evitando así posibles conflictos.

- **Videojuegos con espacios de juego persistentes** (nivel 4): los juegos que pertenecen al último nivel se basan en mapas en los que una gran multitud de jugadores juegan durante un tiempo indefinido, sin basarse en partidas. Cuando los usuarios acceden al juego, deben acceder al mismo mapa y desde el mismo punto que dejaron el juego la última vez, es decir, con los mismos objetos y progreso que tenían, lo cual lleva a que el videojuego deba contener tecnologías que hagan el mundo persistente.

  Conseguir esto supone una gran complejidad a nivel de red, ya que no se pueden perder datos y estos deben estar siempre disponibles para que los usuarios puedan seguir progresando en el juego cuando deseen. Para ello, el videojuego deberá tener un gran número de servidores, cada uno de los cuales tendrá una función específica: inicios de sesión, parches, proxy, uno que deberá contener el mundo del juego que se irá actualizando, otros que serán los encargados de la lógica y de cambiar los estados del juego (los cuáles se enviarán al servidor que contenga el mundo para que este sea actualizado) y otros que se encargarán de la sincronización de los datos, guardándolos en la base de datos y cargándolos de la misma cuando sea necesario enviar esos datos a otros servidores. Este complejo esquema de servidores se puede observar en la **Figura 8**.

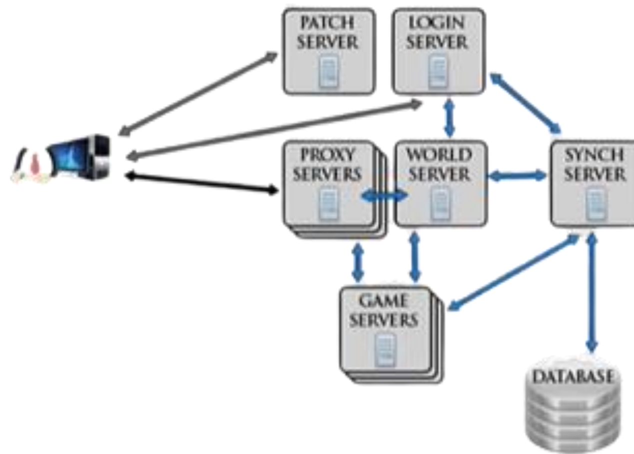


Fig. 8. Esquema de red común en los videojuegos con espacios de juego persistentes [5]

El esquema de red mostrado en la **Figura 8** servirá también para solventar los problemas surgidos en los niveles 1, 2 y 3, ya que tendrán que ser resueltos también en los videojuegos de este nivel. Se puede observar un esquema de red con multitud de servidores en el que el PC del jugador sólo tiene líneas de conexión directas con los servidores de parches/actualizaciones, de inicio de sesión y con el proxy. De esta forma se añade una protección extra al servidor que contiene el mundo del juego, al que se encarga de la sincronización de los usuarios, a los servidores que crean las sesiones de juego y a la base de datos del videojuego, ya que a estos elementos (que son más críticos) sólo podrán acceder otros servidores.

3. Técnicas y procesos de red en los videojuegos

Una vez vistas las distintas formas de clasificar los videojuegos en red y los elementos que pueden estar presentes en la parte red, en este apartado del documento se van a desarrollar algunas de las técnicas y procesos de red más comunes en la industria de los videojuegos. En concreto se van a tratar las técnicas de compensación de latencia, el *matchmaking* y las técnicas antitrampas, siendo técnicas y procesos esenciales para la estabilidad, consistencia, seguridad y jugabilidad de un videojuego en red.

3.1. Técnicas de compensación de latencia

La latencia de red en los videojuegos consiste en el tiempo de respuesta desde que el jugador realiza una acción hasta que esta se ve reflejada en el juego, así como el tiempo que tarda en ver las acciones de los jugadores oponentes desde que estos las han realizado. La latencia puede llegar a empeorar la calidad de experiencia de un juego notablemente, al aumentar los tiempos de respuesta y llevar al juego a una gran inestabilidad al causar un gran “delay” o también llamado retardo [3].

Los videojuegos están basados en frames, es decir, el cambio de estados en un juego, el cual debe ocurrir en intervalos regulares. Para procesar el siguiente frame, se requieren

los mensajes o acciones de los usuarios del frame anterior, con los cuales se avanzará al siguiente estado del juego. Estos mensajes podrán viajar entre los propios usuarios de forma directa (arquitectura de red “peer to peer”) o entre mediante un servidor o *host* de intermediario (arquitectura cliente-servidor con *host* o con servidor dedicado) [4] y vendrán estructurados en forma de paquetes de red. También existe la arquitectura de un sistema distribuido que consiste en un grupo de servidores que se conectan entre sí actuando como pares mientras que los jugadores se conectan a servidores locales. El retardo entre frames es uno de los principales causantes de que se pueda producir una alta latencia en un videojuego en red. Este retardo puede venir dado por problemas cómo una conexión de red deficiente, hardware y/o software defectuoso o incluso una programación incorrecta de la parte red por parte de los desarrolladores del videojuego. Estos problemas pueden llevar a un retardo en el envío y/o recepción de paquetes de red o incluso a la perdida de paquetes, ocasionado problemas de latencia.

En el caso de una arquitectura cliente-servidor con un servidor dedicado, que es la más común en los juegos en red, un servidor será el encargado de guardar y actualizar el estado del juego, mientras que los clientes tendrán una copia del estado del juego. A medida que le lleguen al servidor acciones de los clientes (mediante paquetes), este actualizará el estado del juego y se lo enviará a los clientes (también mediante paquetes de red) para que se les actualice a los jugadores. La latencia de un usuario se medirá en la diferencia de tiempo entre que el jugador realiza una acción y esta se ve reflejada en el videojuego en red siendo renderizado el nuevo mundo con el cambio en el dispositivo del cliente.

La arquitectura peer-to-peer directa o la cliente-servidor con *host*, a pesar de que tiene la ventaja de que no se necesitan ordenadores adicionales a los de los propios jugadores, carece de seguridad ante hackeos, ya que el estado del juego no pasa por un servidor que lo compruebe. Además, en el caos de la arquitectura peer-to-peer directa, los dispositivos de los jugadores requerirán de más recursos hardware al conectarse y transferir directamente a los otros jugadores el estado del juego. Con la arquitectura cliente-servidor con *host* sólo habrá un jugador que necesite utilizar recursos para conectarse con todos los clientes, el cual será el *host* (como ya se ha explicado previamente), lo cual puede conllevar a problemas como bajo rendimiento en el *host*, ventaja de una menor latencia frente al resto de jugadores o problemas de seguridad frente a trampas en el *host* al no haber un servidor dedicado.

Respecto a la arquitectura compuesta por un grupo de servidores, puede suponer una mayor consistencia y escalabilidad para el juego en red, pero a su vez una mayor latencia en casos en los que un jugador que está conectado a un servidor se tenga que comunicar con un jugador que esté en otro servidor distinto. En el proceso de comunicación, primeramente, la acción del jugador pasa a su servidor local, el cual se comunica con el servidor del otro jugador. Después, ambos servidores actualizan el estado del juego y se lo envían a los jugadores para que se actualicen las copias del estado del juego en los clientes [3].

La banda ancha de red de cada jugador, la distancia entre los nodos de red del juego (jugadores y servidores) y la propia programación de la parte red del juego, son factores claves de la latencia de red. Los problemas que puede llegar a causar la latencia son solucionados por los programadores de videojuegos mediante las llamadas técnicas de compensación de latencia, que intentarán reducirla al máximo para proporcionar una buena experiencia de juego a los jugadores en los videojuegos en red [3].

También existe la llamada latencia local, que dependerá de los dispositivos de entrada, el hardware y software del dispositivo, sistema operativo y el resto del desarrollo del juego que está fuera de la programación de la parte red. En los videojuegos en red, existe tanto latencia en red como latencia local y que, como ya se ha comentado, consistirán en el retraso del envío y/o entrega de paquetes de red, así como incluso en la pérdida de algunos de ellos.

Las técnicas de compensación de latencia empleadas dependerán principalmente del tipo de juego en el que se van a aplicar y de las acciones y el estado del jugador, ya que en algunos casos será más importante el tiempo de respuesta (lo que tarda el estado del juego en actualizarse desde que el jugador ha realizado una acción) y en otros se dará una mayor prioridad a la consistencia (diferencia entre el estado del juego del mundo del servidor y las copias del estado del juego renderizadas en los mundos de los clientes). En un videojuego en red lo óptimo es tener el menor tiempo de respuesta y la mayor consistencia posibles. Habrá casos en los que estas técnicas aumenten la consistencia a cambio de aumentar el tiempo de respuesta y casos en los que suceda lo contrario, el tiempo de respuesta se reducirá a cambio de reducir también la consistencia del juego [3].

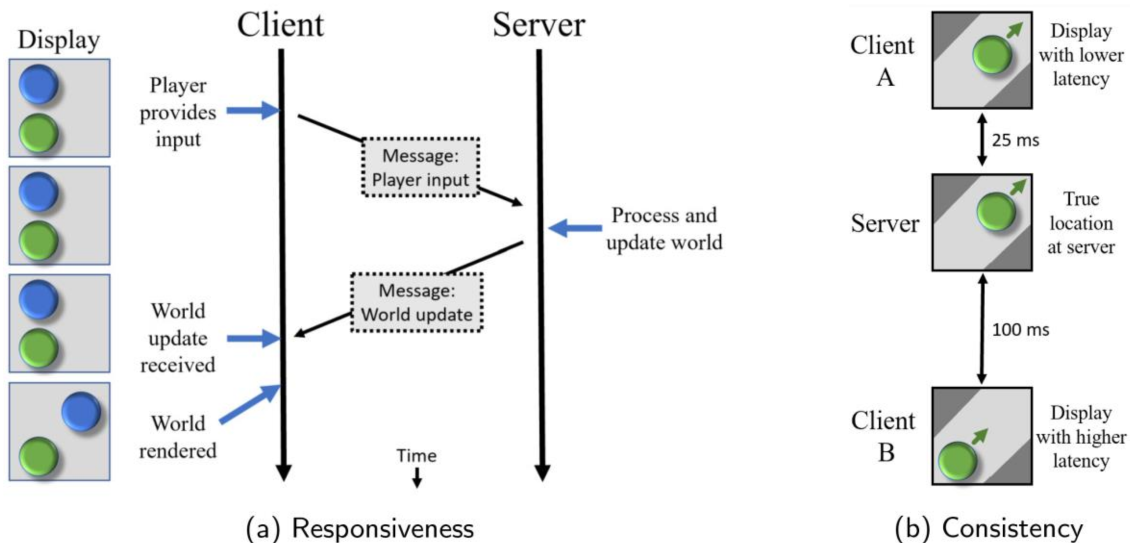


Fig. 9. Tiempo de respuesta y consistencia en un videojuego en red con arquitectura cliente-servidor [3]

En la **Figura 9** se puede observar cómo en la parte b de la imagen, que muestra un ejemplo de consistencia, el cliente A tiene una mayor consistencia que el B al tener una menor latencia, lo cual conlleva a que el mundo renderizado en el cliente A sea muy similar al que existe en el estado del juego original guardado en el servidor. El mundo renderizado en el cliente B, al tener una gran latencia, tendrá una gran diferencia con respecto al del cliente A, lo cual lleva a una baja consistencia del juego en red, ya que ambos jugadores no ven lo mismo en su pantalla simultáneamente [3].

Respecto a los tiempos de latencia aceptables, dependerán del tipo de juego cómo se ha comentado previamente, ya que, por ejemplo, en los juegos en los que se requiera de una alta precisión (shooters, juegos deportivos o de acción) las latencias no deben superar los 100 ms para que el juego sea jugable (e incluso los 50 ms dependiendo del juego), mientras que en juegos cómo los RTS o los RPG que no requieren de tanta precisión, pueden ser aceptables latencias de centenares de milisegundos. En los juegos basados en turnos, incluso una latencia de un par de segundos no afectará a la experiencia de los jugadores. General y especialmente en los juegos en los que se requiere de una gran

precisión y reflejos, una menor latencia de red supondrá siempre una mejor calidad de experiencia y un mayor rendimiento para los jugadores [3].

Las técnicas de compensación de latencia pueden dividirse en 4 grupos entre los cuales existen 11 técnicas. En la **Figura 10** se puede observar esta división en forma de un árbol jerárquico de arriba hacia abajo.

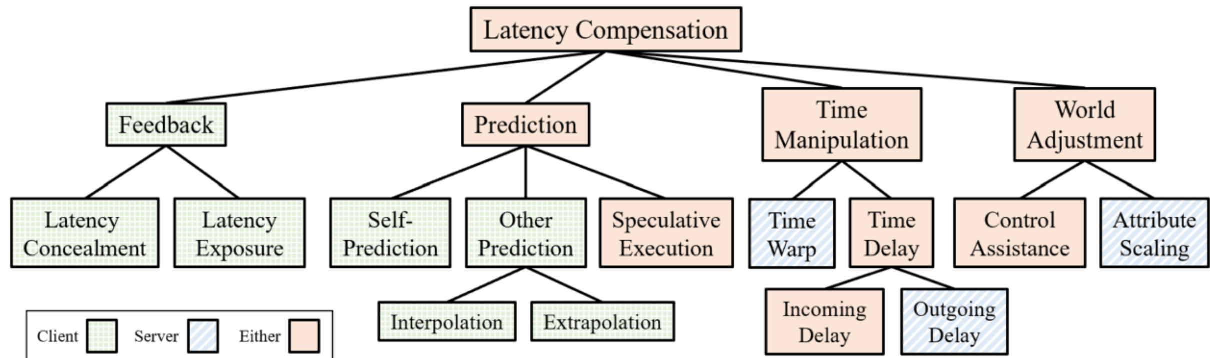


Fig. 10. Árbol jerárquico de las técnicas de compensación de latencia en los videojuegos en red [3]

Como se puede ver en la **Figura 10**, las técnicas marcadas en verde están enfocadas a la parte del cliente, mientras que las azules van dirigidas al servidor y las rojas pueden implicar a ambas partes [3]. Por tanto, la división de las técnicas de compensación de latencia quedaría marcada por cuatro grupos, los cuales se van a desarrollar en los siguientes subapartados.

3.1.1. Técnicas de compensación de latencia basadas en feedback

Las técnicas de compensación de latencia basadas en feedback dan información visual o audible al usuario, sin cambiar el estado del mundo del juego. Existen 2 técnicas [3]:

- **Ocultación de la latencia**: esta técnica se basa en ocultar la latencia a los jugadores a través de efectos visuales. Cuando el jugador realiza una acción, el juego renderizará efectos visuales sin modificar la posición del jugador hasta que reciba la confirmación del cambio del estado del juego por parte del servidor, creando así en el videojuego una sensación de un tiempo de respuesta muy reducido, lo cual mejorará la calidad de experiencia de juego para los usuarios.

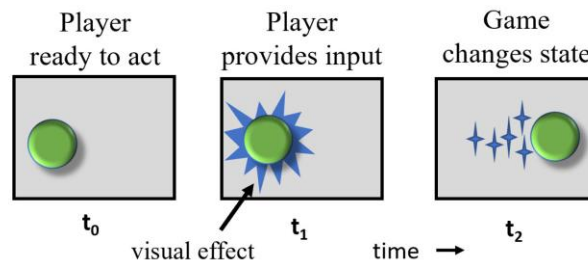


Fig. 11. Representación de la técnica de ocultación de la latencia [3]

En la **Figura 11** se pueden observar tres estados: en el t0 el jugador está listo para actuar, en el t1 el jugador realiza una acción y en el t2 se muestra que ha habido un efecto visual de partículas para ocultar la latencia hasta que se ha aplicado la acción en el estado del juego.

- **Exposición de la latencia**: esta técnica muestra la latencia a los jugadores, pudiendo realizarse esto de diversas formas: mostrar los ms de latencia en la esquina de la pantalla, mostrar mediante sombras o siluetas difuminadas una aproximación de dónde debería estar el personaje u objetos en el servidor (la actualización de la posición o acción del personaje no ha llegado del servidor al cliente todavía), rejillas cubriendo los objetos afectados por la latencia, mediante gráficos de barras… Se ha comprobado que esta técnica sólo reduce notablemente la latencia cuando esta es muy alta (a partir de 800 ms).

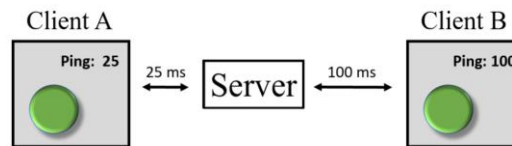


Fig. 12. Escenario en el que se puede aplicar la técnica de exposición de latencia [3]

En la **Figura 12** se muestra un escenario en el que hay dos clientes que tienen diferencias grandes de latencia con el servidor (25ms vs 100 ms), por lo que es un escenario ideal para aplicar la técnica de exposición de latencia.

3.1.2. Técnicas de compensación de latencia basadas en predicción

Las técnicas de compensación de latencia basadas en predicción renderizan cambios en el mundo de la parte del cliente prediciendo el siguiente estado del juego que habrá en la copia original del estado del juego, alojada en el servidor. En caso de que la predicción no coincida con la actualización real del estado del juego en el servidor, se generará una inconsistencia y cuándo el servidor le envíe la actualización al cliente, el mundo virtual del cliente tendrá un cambio visual brusco que el jugador notará. Existen 4 técnicas [3]:

- **Auto predicción**: esta técnica es utilizada en casos en los que la parte cliente del videojuego tiene los recursos necesarios para que cuando cuándo el usuario realiza una acción, se pueda predecir y renderizar el siguiente estado del juego antes de recibir la respuesta del servidor del juego. Para ello se necesita que la parte cliente del juego incluya un motor de videojuegos totalmente funcional, para ser capaz de renderizar el siguiente estado del juego por sí mismo. Cuando llega la respuesta del servidor, en caso de que esta no coincida con la predicción (porque el cliente no tenga incluido un objeto que sí está en el servidor y que impide la acción del jugador, por ejemplo), se deberá reajustar el mundo del jugador para eliminar la inconsistencia.

- **Interpolación**: esta técnica se utiliza para predecir un estado anterior del juego de otros personajes u objetos ajenos al del propio jugador (incluyendo NPCs controlados por una inteligencia artificial), basándose en el estado actual y en otros estados anteriores al que se quiere predecir. Se utiliza para que el juego se vea fluido visualmente y no haya cambios visuales bruscos que puedan arruinar la calidad de experiencia del juego para los usuarios. Se suele utilizar en videojuegos en red en los que el mundo de la parte cliente del juego se actualiza y renderiza con más frecuencia de la que el servidor envía actualizaciones a los clientes.

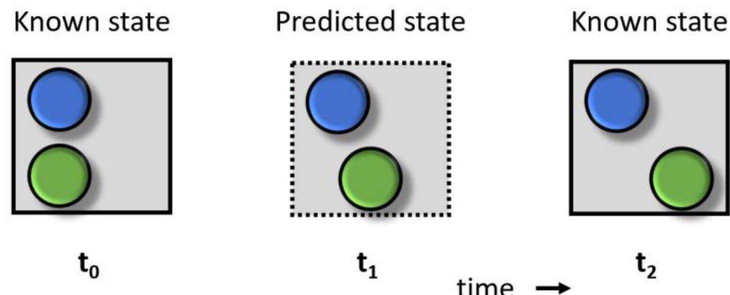


Fig. 13. Representación de la técnica de interpolación [3]

En la **Figura 13** se pueden observar tres estados presentes en un cliente de un videojuego: el t0 es el estado de juego conocido, el t2 es el siguiente estado de juego conocido y el t1 es el estado que se predice aplicando la interpolación para que el juego se vea más fluido.

- **Extrapolación**: esta técnica se utiliza para predecir un estado futuro del juego de otros personajes u objetos ajenos al del propio jugador (incluyendo NPCs controlados por una inteligencia artificial) basándose en el estado actual del estado del juego y las características actuales de ese personaje u objeto, cómo, por ejemplo, la velocidad a la que va el personaje. Existen otras características más avanzadas, cómo la aceleración, masa, fricción, orientación… También se tendrán en cuenta características del mapa del juego, cómo el clima, los caminos disponibles o el destino. Teniendo en cuenta todo esto, se calculará la nueva posición mediante algoritmos de predicción, cuya fórmula dependerá del juego y todas las características que haya que tener en cuenta. Dos ejemplos simples de algoritmos de predicción son:
  - Algoritmo basado en velocidad constante: *pt1 = pt0 +vt0 (t1 − t0).*
  - Algoritmo basado en una aceleración constante: *pt1 = pt0 +vt0 (t1 − t0) + ½ a (t1 − t0)^2.*

En las 2 fórmulas anteriores las “p” representan la posición, las “v” la velocidad y las “t” el tiempo. Una vez calculada la nueva posición, se renderizará el mundo de la parte cliente con la actualización. El servidor, comparará la posición predicha con la posición final real y si la diferencia supera un cierto umbral, le enviará un mensaje al cliente para que cambie la posición de ese personaje u objeto, actualizando también sus características.

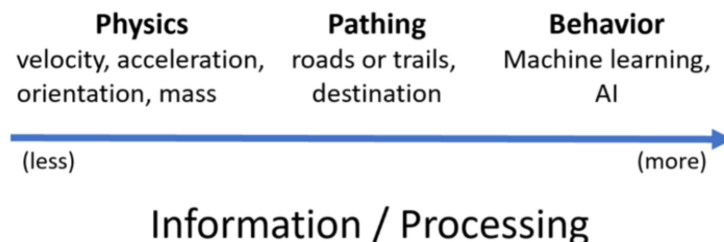


Fig. 14. Factores para la predicción en la técnica de extrapolación [3]

En la **Figura 14** puede ver un resumen de como se lleva a cabo la predicción en la extrapolación. En la primera fase, se tendrán en cuenta las características del personaje u objeto, en la segunda las características del ambiente en el mapa y, por último, se tendrá también en cuenta el comportamiento del personaje en

estados pasados del juego, pudiendo también aplicar técnicas más complejas de machine learning o IA.

Esta técnica de compensación de latencia es una de las más utilizadas ya que puede llegar a reducir la latencia en hasta 10 veces, dando una mayor calidad de experiencia de juego y haciendo el juego más equilibrado para todos los jugadores al no haber tanta diferencia de latencia entre ellos. Reduce la latencia considerablemente en casi todos los casos, haya 300 o 100 ms de latencia originalmente, por lo que es muy útil para los programadores de videojuegos en red. También reduce bastante el "bitrate" (tasa de bits enviados), ya que, si la predicción realizada mediante la extrapolación no se excede del umbral y, por tanto, se considera correcta, el servidor no necesitará enviar un mensaje de actualización del estado del juego al cliente.

- **Ejecución especulativa**: la ejecución especulativa consiste en el preprocesamiento de la posible siguiente acción del jugador, para que cuando este la lleve a cabo, si coincide con una de las acciones preprocesadas, no haga falta esperar a la confirmación del servidor a esa acción para poder renderizar el mundo con la actualización correspondiente. En la etapa de preprocesamiento se predecirán varias acciones y una vez el jugador realiza la acción, se descartarán todas las que no coinciden (si no coincide ninguna será necesario esperar a la respuesta del servidor con la actualización del mundo tras esa acción). El preprocesamiento se basará principalmente en acciones previas.

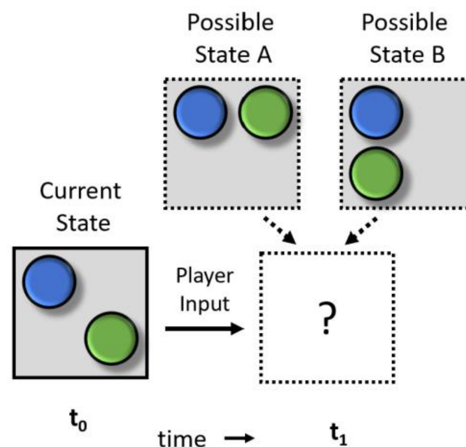


Fig. 15. Representación de la técnica de ejecución especulativa [3]

En la **Figura 15** se pueden ver dos estados: t0 es el estado de juego actual y t1 el siguiente estado, el cual contiene dos posibles estados preprocesados.

Esta técnica reducirá la latencia notablemente sólo en las ocasiones en las que el jugador tenga una gran latencia (a partir de 250 ms), aumentando el rendimiento y la calidad de experiencia de juego del usuario. Sin embargo, gracias a diversas pruebas y evaluaciones, se ha podido comprobar que para latencias de entre 0 y 128 ms la mejora al aplicar esta técnica es insignificante.

3.1.3. Técnicas de compensación de latencia basadas en la manipulación del tiempo

Este tipo de técnicas de compensación de latencia alteran el tiempo virtual (el que ocurre en el mundo del juego) para procesar estados del juego o solucionar problemas con las acciones de los jugadores. Existen 3 técnicas [3]:

- **Deformación temporal**: esta técnica se basa en retroceder en el servidor a un estado del juego de la parte cliente que se ha perdido debido a la latencia y que incluye una acción relevante que debe ser actualizada en el estado del juego del mundo del servidor.

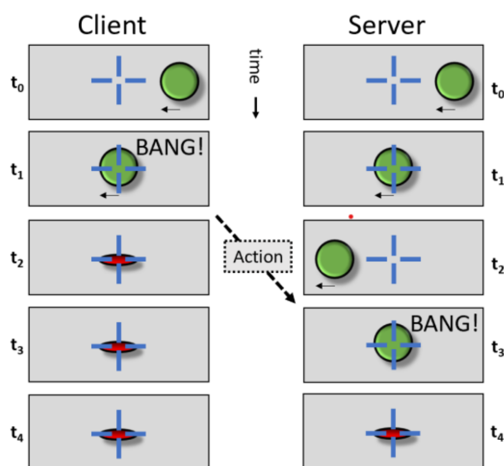


Fig. 16. Ejemplo de uso de la técnica de deformación temporal [3]

En la **Figura 16** se puede ver un ejemplo del uso de la técnica de deformación temporal en un FPS (shooter en primera persona) en el que el punto verde es un jugador oponente y la cruceta es la mira del arma del usuario. En la parte del juego del cliente se puede ver cómo el usuario dispara e impacta en el jugador contrario en el momento t1, mientras que, en el servidor, debido a la latencia, la acción de disparo del usuario no llega a tiempo y se tiene que recuperar la acción del cliente en el momento t4 del servidor. Después, se actualizará el estado del juego en el servidor, el cual se enviará al resto de clientes.

Esta técnica de compensación de latencia puede llevar a inconsistencias en el juego al poder suponer cambios bruscos visuales para los jugadores oponentes afectados (en el caso del ejemplo al punto verde), ya que se pueden ver con situaciones en las que sea heridos o muertos tras haber llegado a una zona segura del mapa en la que es imposible que haya recibido un disparo (ya que el disparo realmente lo ha recibido antes de llegar ahí en la parte del juego del otro cliente, realizando el posterior retroceso en el servidor).

Esta técnica sirve también para solucionar problemas generados por las técnicas de compensación de latencia basadas en predicción, ya que pueden recuperar acciones perdidas en la fase de predicción. La deformación temporal es útil principalmente en los mencionados FPS y en los juegos arcade, aumentado la precisión y las puntuaciones en partidas, a costa de algunos casos de inconsistencia en las partidas, aunque generalmente compensa, ya que es más grande el porcentaje de aumento en precisión y puntuaciones que el de aumento en casos visuales de inconsistencia.

- **Retardo de entrada**: esta técnica de compensación de latencia es utilizada para conseguir que los juegos sean más justos y los estados de juego más consistentes, ya que trata de igualar las latencias de todos los jugadores. Para ello, aplicará un proceso de retardo de forma local en el que cuando el jugador realice una acción, los efectos de esta no se vean reflejados en el mundo de ese cliente hasta que esta acción no haya llegado a los otros clientes, para lo cual primero deberá haber pasado por el servidor del videojuego. Es decir, al jugador que realiza la acción se le aplicará un retardo local que será igual a la suma de la latencia entre ese cliente y el servidor y la máxima latencia entre el servidor y el resto de los clientes. Por lo tanto, teniendo en cuenta la máxima latencia entre el servidor y los clientes, se realiza un proceso de sincronización en el que a todos los mundos de los clientes se les renderizará a la vez el mundo con la nueva actualización provocada por la acción.

  Esta técnica elimina la inconsistencia de los estados de jugo al aplicarse las actualizaciones en todos los clientes a la vez, aunque a cambio aumentará considerablemente el tiempo de respuesta a los jugadores con latencias bajas, ya que dependerán de los jugadores con latencias altas para que sus acciones se apliquen.

  El retardo de entrada se puede combinar con técnicas de predicción y de ocultación de latencia para mejorar el rendimiento y la calidad de experiencia al crear una sensación visual de menores tiempos de respuesta. El aumento del tiempo de respuesta al aplicar la técnica de retardo de entrada se notará especialmente en los juegos FPS, ya que son juegos en los que se requieren de acciones rápidas, con precisión y reflejos.

- **Retardo de salida**: esta técnica se utiliza para que los videojuegos multijugador sean más justos mediante el equilibrio de las latencias por parte del servidor. Este proceso lo lleva a cabo el servidor del juego mediante la aplicación de retardos en el envío de sus mensajes a los clientes, teniendo en cuenta la máxima latencia entre el servidor y los clientes (aunque existirá un rango máximo de retardo que no se puede sobrepasar).

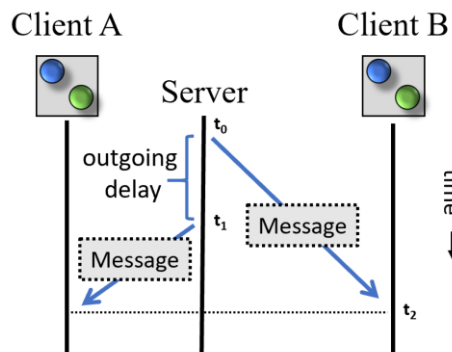


Fig. 17. Ejemplo de uso de la técnica de retardo de salida [3]

En la **Figura 17** se observa un ejemplo de aplicación de la técnica de retardo de salida, donde se le aplicará un mayor retardo de salida al cliente A al tener una menor latencia con el servidor que el cliente B (el retardo será siempre inversamente proporcional a la latencia). De esta forma, los mensajes del servidor llegarán a la vez a los clientes y los estados de juego de los clientes se actualizarán

a la vez, haciendo que el juego sea más justo, especialmente en los juegos en los que se requiera de gran precisión y reflejos, cómo los FPS.

3.1.4. Técnicas de compensación de latencia basadas en el ajuste del mundo del juego

Este tipo de técnicas de compensación de latencia se basan en modificar los estados del juego para compensar las diferencias de latencia entre los distintos jugadores. Existen 2 técnicas [3]:

- **Asistencia de control**: esta técnica modifica el resultado de las acciones de los jugadores para que estas se acerquen más a los objetivos y aumenten el rendimiento y las puntuaciones de los usuarios en los videojuegos en red. La asistencia se aplicará en proporción a la latencia del jugador, siendo aplicada una mayor asistencia a los jugadores con mayor latencia, para reducir así la desigualdad, al ser más complicado competir con más latencia, especialmente en los juegos que requieran de una gran precisión.

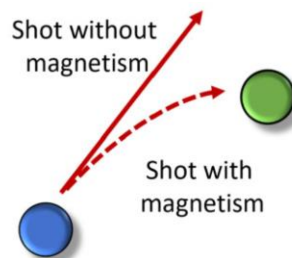


Fig. 18. Ejemplo de uso de la técnica de asistencia de control [3]

En la **Figura 18** se puede ver un ejemplo de la aplicación de la técnica de asistencia de control en un juego de género "shooter", en el que el jugador azul apunta y dispara al verde, siendo su disparo desviado por la técnica de asistencia de control e impactando en el jugador verde, compensando así la latencia.

Esta técnica es especialmente útil para aumentar el rendimiento de los jugadores en los juegos "shooter", dónde se requiere de una gran precisión, aumentando considerablemente la puntuación y la calidad de experiencia de juego en los jugadores con una gran latencia.

- **Escalado de atributos**: esta técnica modifica las características de los personajes o de los elementos del mundo virtual en base a la latencia que tenga cada jugador. El objetivo es equilibrar la dificultad para todos los jugadores, mejorando para los jugadores con una mayor latencia las características del personaje (velocidad, fuerza, tamaño, etc.) y/o modificando los obstáculos del mapa para reducir la dificultad que supone jugar con una latencia grande. También se puede aumentar el tiempo que los jugadores tienen para realizar una acción determinada en caso de tener una gran latencia, para no reducir el rendimiento.

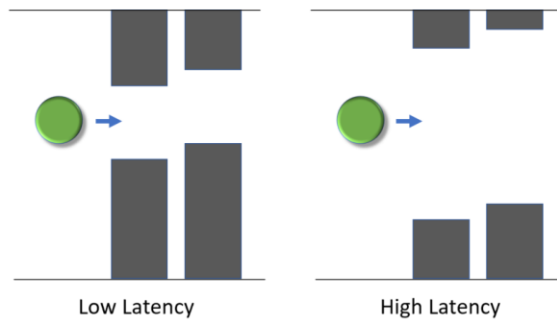


Fig. 19. Ejemplo de uso de la técnica de escalado de atributos [3]

En la **Figura 19** se puede observar un ejemplo de uso de esta técnica en el que el jugador verde debe avanzar esquivando los obstáculos del mapa. En el ejemplo izquierdo, el jugador tiene poca latencia y en el derecho una gran latencia, por lo que se reduce el tamaño de los obstáculos para compensar la latencia del jugador de la derecha.

Esta técnica es útil generalmente en todos los videojuegos en red, ya que permite equilibrar las partidas al poder adaptar el juego a la latencia del usuario, quitando la ventaja que tienen los jugadores con poca latencia con respecto a los que tienen una gran latencia. El uso del escalado de atributos también aumenta notablemente la calidad de experiencia de juego, al no depender tanto de la latencia.

En la industria comercial de videojuegos en red, la mayoría de los videojuegos son de código privado, por lo que no se puede saber que técnicas de compensación de latencia utilizan, aunque en la mayoría de los juegos conocidos se utilizan para mejorar el rendimiento de los jugadores y mejorar la calidad de experiencia de los juegos. Debido a diversos juegos de código abierto y las publicaciones de contenido técnico de desarrolladores de videojuegos, se puede establecer que las técnicas de compensación de latencia más utilizadas son la extrapolación y la deformación temporal (utilizadas en prácticamente cualquier género), aunque también son utilizadas las de retardo de entrada, auto predicción, interpolación y exposición de la latencia. Las técnicas de compensación de "feedback" y de ajuste del mundo necesitan una mayor investigación y desarrollo para poder ser utilizadas y sacar su potencial. El género del que hay más constancia de que se utilizan técnicas de compensación de latencia es el FPS (utilizando diversos tipos de técnicas), ya que es un tipo de juego que requiere de una gran precisión y un tiempo de respuesta muy pequeño, pero también se utilizan en el resto de los géneros de videojuegos en red [3].

3.2. Matchmaking

El *matchmaking* es un proceso presente en los videojuegos en red mediante el cual los jugadores (clientes) son asignados a una partida en específico junto a otros jugadores pudiendo basarse en distintos factores como el mapa, el nivel de habilidad, ubicación del cliente y el servidor, etc.

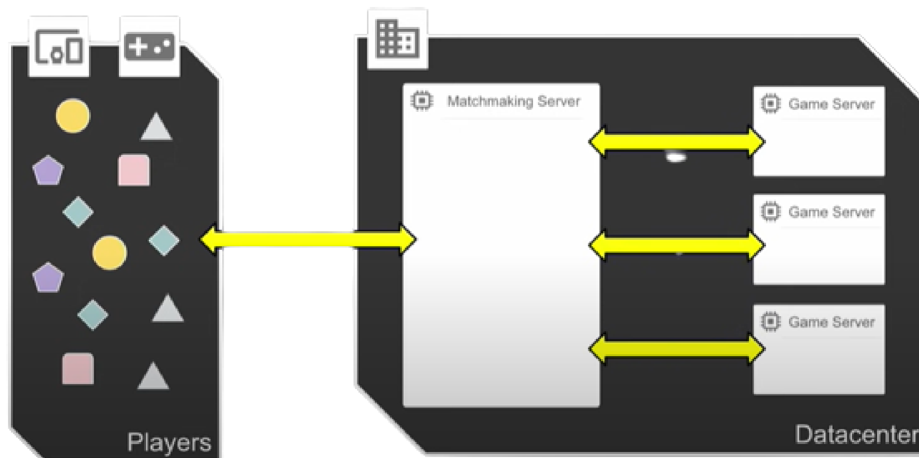


Fig. 20. Arquitectura común en un sistema de *matchmaking* [6]

En la **Figura 20** se puede ver una representación común en los sitemas de *matchmaking* en los videojuegos en red. En la parte iziquierda se encuentran los dispositivos de todos los jugadores que están solicitando una partida en el videojuego. De intermediario, se encuentra el denomiando como servidor de *matchmaking*, que será el encargado de dividir a todos los jugadores en distintas partidas en base a ciertos factores como los que se han mencionado, cada una de las cuales estará alojada en un servidor de juego dedicado distinto.

Dentro del motor de videojuegos de Unity existe una alternativa de *matchmaking* más desarrollada [6], la cual se puede resumir en la **Figura 21**.

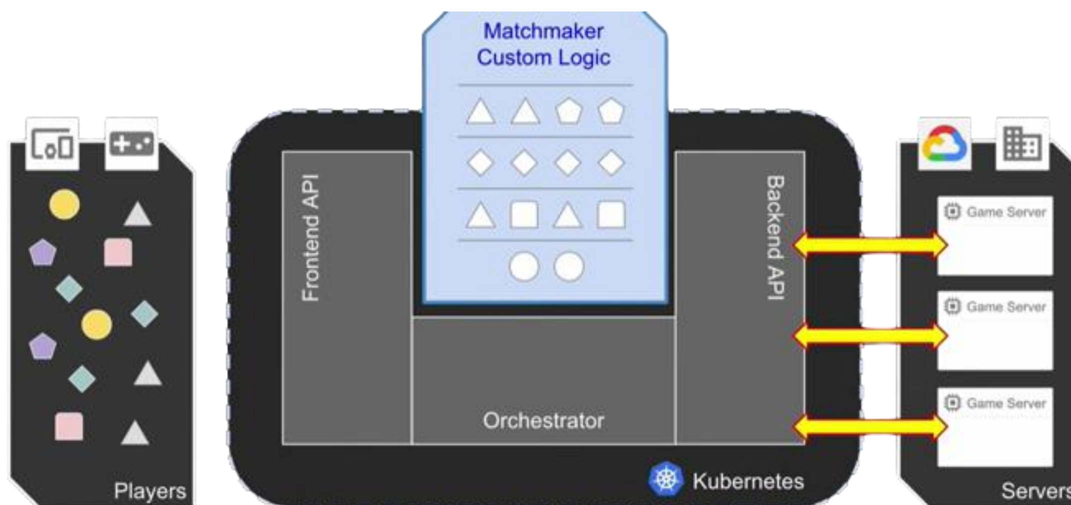


Fig. 21. Arquitectura alternativa de *matchmaking* propuesta por Unity [6]

En el caso de la **Figura 21**, la parte intermediaria está dentro de un contenedor “Kubernetes” de la nube de Google Cloud, el cual está compuesto por una lógica personalizada creada por el desarrollador junto a otra parte implementada por defecto por Unity. La parte de Unity está compuesta por una API de *frontend* que recibirá las peticiones de partida de los jugadores y se las pasará al orquestador, donde teniendo en cuenta la logica personalizada implementada por el desarrollador enviará a la API de *backend* –(implementada por Unity) la organización de los jugadores en las distintas partidas. Por último, la API de *backend* conectará a los jugadores a los distintos servidores de juego dedicados según las indicaciones dadas por el orquestador (los cuales pueden

ser propiedad de los desarrolladores del videojuego o servidores ubicados en la nube de Google Cloud).

En resumen, el proceso usual de *matchmaking* se puede dividir en 5 fases principales:

- El jugador (cliente) busca partida en el videojuego en red (pudiendo introducir ciertos filtros de búsqueda o no), lo cual recibe el sistema de *matchmaking* implementado en el videojuego en red.
- Esperar a que un número X de jugadores con los mismos requisitos de partida (el mapa o modo de juego en específico que se haya seleeionado o ciertos factores que no dependan de los filtros seleccionados como la ubicación geográfica del jugador o su habilidad) estén buscando partida. Ese número X deberá ser establecido y será el número mínimo de jugadores en una partida.
- Esperar un número de segundos establecido a que se unan más jugadores. También habrá que establecer un número máximo de jugadores en una partida, el cual dependerá entre otras cosas de la potencia de los servidores del videojuego en red. Si se llega a ese número de jugadores máximos, se terminará el periodo de espera. Si no se llega, cuándo acabe el número de segundos establecido, la partida empezará si se ha alcanzado el número mínimo de jugadores.
- Preparar un servidor para la partida.
- Conectar a todos los jugadores al servidor en cuestión para dar comienzo a la partida en el videojuego en red.

En caso de que no se llegue al número de jugadores mínimo para una partida antes del tiempo que se haya establecido, los usuarios serán sacados de la sala de espera y se les notificará lo ocurrido, pudiendo realizar una nueva búsqueda de partida con las mismas caraterísticas (mapa, modo de juego,etc.) u otras con las que pueda resultar más sencillo encontrar jugadores (mapas o modos de juego más populares).

El *matchmaking* en los videojuegos se suele basar en su mayoría en los niveles de habilidad de los jugadores, tratando de esta forma que las partidas sean lo más equilibradas posibles al emparejar jugadores con un nivel de habilidad similar, el cual se basa en una puntuación específica. Sin embargo, también se puede basar a su vez en otros factores cómo la localiación de los jugadores o la calidad de la conexión, para refinar aún más los emparejamientos.

El principal factor en el que se basa el *matchaming* en los videojuegos en red es la habilidad, la cual se puede medir de diversas formas mediante sistemas de clasificación. A continuación, se van a explicar los principale sistemas de clasificación de habilidad que se pueden encontrar en los videojuegos en red:

- **Elo**: este sistema de calificación fue creado por Arpad Elo [10] (un profesor de física estadounidense nacido en Hungría) para utilizarse en ajedrez, aunque a día de hoy, se ha adaptado para también poder ser utilizado en otros juegos. En los videojuegos en red se utilizan variaciones de este sistema o este sistema cómo complemento de otros. En el sistema Elo cada jugador está representado por una puntuación, y cuanto más alta sea esta, más habilidoso será el jugador.

  El objetivo principal de las clasificaciones de Elo es predecir con precisión los resultados de las partidas, para lo cual se puede utilizar la siguiente fórmula, en la

que E_A es la probabilidad de que gane el jugador A al B en una partida (midiéndose entre 0 y 1):

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}.$$

R_A representa la puntuación actual del jugador A y R_B la del B. El cálculo de la probabilidad de victoria del jugador B se hace utilizando la misma fórmula, pero cambiando de lugar R_A por R_B. También podría ser calculada restándole a 1 el E_A calculado previamente.

La puntuación de cada jugador debe actualizarse cada un número determinado de partidas para que represente de la manera más precisa posible la habilidad del jugador o equipo. Para ello se utilizará la siguiente fórmula, con la que nos basaremos en que la puntuación se actualiza cada 5 partidas:

$$R'_A = R_A + K(S_A - E_A).$$

Para calcular la nueva puntuación del jugador (R'_A), se necesitará utilizar la puntuación anterior a las partidas (R_A), la suma de los resultados de las partidas (S_A) y la suma de los resultados esperados calculados con la primera fórmula (E_A). Para calcular S_A se tendrá en cuenta que una derrota cuenta 0, un empate 0.5 y una victoria 1. E_A será la suma de las probabilidades de victoria de las 5 partidas, para cuyo cálculo se utilizará la primera fórmula, cómo se ha comentado previamente. El factor K se deberá definir en base a cómo de grande se desea que sean los cambios en las actualizaciones, ya que cuánto más grande sea K, más grandes serán los cambios. En ajedrez se suele utilizar $K=16$ para los jugadores maestros y $K=32$ para los principiantes, ya que estos han jugado menos partidos y, por tanto, no se tienen tantos datos de partidas sobre ellos.

- **Glicko**: este sistema de clasificación fue creado por Marck Glickman [11] con el propósito de mejorar el sistema Elo y ser utilizado en ajedrez, al igual que el Elo. Sin embargo, con los años se ha adaptado par ser utilizado también en otro tipo de juegos. Para mejorar el sistema Elo, introduce la denominada desviación de calificaciones (llamada *RD*, de *Rating Deviation*), la cual mide la precisión con la que la puntuación representa el nivel de habilidad de un jugador (a menor desviación mayor precisión en el cálculo de la puntuación) y es equivalente a una desviación estandar. Esta desviación será mayor cuánto más tiempo lleve un jugador sin jugar (ya que su nivel de habilidad puede haber cambiado) o cuántas menos partidos haya jugado (al tener pocos datos de las partidas del jugador). Tanto la desviación cómo la puntuación se irán actualizando cada partida y cuánto mayor sea la desviación, mayor será también el cambio de puntuación en cada actualización. Además, con este sistema, la puntuación de los jugadores se representa en un intervalo de confianza (para una puntuación de 1500 y un intervalo de confianza del 95%, la puntuación del jugador estaría representada por un intervalo entre 1400 y 1600).

  Por tanto, la principal diferencia entre el sistema Glicko y el Elo, es la desaparición de la constante K, la cuál es sustituida por la RD, mejorando de esta manera el sistema al porporcionar una mayor precision en la medición del nivel de habilidad de los jugadores.

Posteriormente apareció el sistema de clasificación Glicko 2, una versión mejorada que trajo cómo novedad el conecpto de la volatilidad de la puntuación [12]. La volatilidad mide el grado de fluctuación esperada en la puntuación de un jugador. Por ejemplo, si un jugador gana muchas partidas seguidas (más que la media de su histórico), la volatilidad será alta, mientras que si tiene un periodo de estabilidad de resultados (se acerca a la media de su histórico de resultados), la volatilidad será baja al no haber una gran varación en las actualizaciones de su puntuación.

- **TrueSkill**: este sistema de clasificación basado en el Glicko es uno de los más utilizados en la actualidad de los videojuegos en red, creado por Microsoft y siendo utilizado por Xbox Live en sus juegos [13]. Este sistema se enfoca principalmente en los videojuegos con partidas por equipos, sin depender del número de jugadores en cada uno de ellos [14]. Para ello, utiliza un sistema de inferencia bayesiana, representando el nivel de habilida de un jugador mediante una distribuión normal o Gaussiana N, en la que μ es la media estimada del nivel de habilidad del jugador (Microsoft la utiliza entre 0 y 50), y σ es el nivel de confianza en esa estimación. Dicho esto, $N(x)$ podría interpretarse cómo la probabilidad de que la puntuación de habilidad de un jugador sea x. Para que el nivel de confianza de la puntuación de habilidad de un jugador sea elevado, el jugador deberá haber jugado un mínimo de 20 partidas.

  Es importante recalcar que este sistema está patentado por Microsoft y, por tanto, no es de código abierto y es exclusivo de Microsoft, aunque existen alternativas de código abierto que intentan imitarlo.

  Este sistema permite mejorar considerablemente la distribuación de las puntuaciones de los jugadores, consiguiendo que los emeparejamientos sean más equilibrados que con los sistemas de clasificación Elo y Glicko.

  En 2018, se lanzó una actualización, denominada cómo TrueSkill 2 y que mejora notablemente las prestaciones del sistema de clasificación.

3.3. Técnicas antitrampas

Las trampas en los videojuegos tienen un papel importante, ya que pueden generar grandes desigualdades entre los jugadores si no se consiguen impedir, afectando gravemente a la experiencia de juego de los usuarios. Si un usuario consigue hacer trampas en un videojuego en red, este tendrá ventajas que el resto de los usuarios no tienen, pudiendo arruinar un videojuego al generar malestar entre los usuarios por no jugar en las mismas condiciones.

Para impedir esto, los desarrolladores de videojuegos implementan sistemas antitrampas, que deberán detectar los intentos de hacer trampas para evitar que estos tengan efecto real en el videojuego, pudiendo también penalizar al usuario que lo ha intentado para que no lo vuelva a hacer. El sistema antitrampas deberá actualizarse regularmente para pulir posibles fallos que permitan las trampas, por lo que es una parte clave en el desarrollo de un videojuego en red.

Antes de nada, se van a explicar los distintos tipos de trampas en los vieojuegos en red, los cuales se dividirán en trampas leves y graves [15]:

- **Trampas leves**: son trampas realizadas a través de las propias mecánicas del juego, las cuales se utilizan de una forma inesperada por los desarrolladores del videojuego para obtener ventajas. Existen dos tipos:

    - Bugs o errores: los bugs o errores son comportamientos inesperados de alguna mecánica del videojuego, los cuales pueden ser aprovechados por el usuario para obtener algún beneficio o ventaja sin necesidad de utilizar ningún tipo de programa o herramienta. Estos bugs deben ser corregidos por los desarrolladores mediante actualizaciones en las que modifiquen el código del videojuego.

    - Compra de bienes o servicios del juego a terceros con dinero real: en muchos videojuegos en red existe la posibilidad de realizar compras a terceros fuera del videojuego, para adquirir objetos del juego, personajes, cuentas con niveles muy altos, etc. Esto aporta una ventaja con respecto al resto de jugadores, ya que permite adquirir ciertas cosas sin invertir el tiempo necesario para conseguirlas. Para detectar este tipo de trampas, se puede utilizar la geolocalización de las acciones en el juego y la frecuencia de inicios de sesión para que automáticamente detecte actividades inusuales y puedan ser investigadas y penalizadas.

- **Trampas graves**: son trampas realizadas a través de programas externos que pueden modificar el código del juego o alterarlo mediante el envío de paquetes modificados. Existen varios tipos:

    - Bots y herramientas automatizadas: este tipo de trampas utiliza programas externos que simulan la actividad del usuario para conseguir progresar en el videojuego sin necesidad de que el jugador actúe. El videojuego deberá estar programada para reconocer patrones inusuales de acciones para evitar este tipo de trampas.

    - Utilizar datos secretos del juego: uso de programas para obtener datos secretos de la memoria de la parte cliente del videojuego o para interceptar datos de los propios paquetes de red. Para evitar este tipo de trampas, el desarrollador de un videojuego debe tener cuidado con la información que se guarda en la memoria de la parte cliente, debiendo minimizarla.

    - Modificación de paquetes y ataques de reproducción: este tipo de trampas utiliza programas externos para enviar paquetes de red modificados al servidor del videojuego, haciéndole llegar así acciones que en realidad el jugador no ha realizado. Este tipo de herramientas también se utilizan para llevar a cabo ataques de reproducción, en los que se envía múltiples veces un paquete que implique una acción específica, como, por ejemplo, el envío de un disparo (al enviar ese paquete múltiples veces puede acabar con la vida de otro jugador, aunque no le haya disparado). El desarrollador del videojuego deberá programar correctamente la parte del servidor para no permitir la recepción de este tipo de paquetes.

- Suplantación de identidad: este tipo de trampas se basa en modificar los datos de un paquete de red para hacerse pasar por otro jugador. Esto puede suponer, por ejemplo, que al enviar un paquete en el que se indica que otro jugador debe suicidarse, ese jugador quedará eliminado sin que se tenga que hacer nada. También se puede aplicar a casos cómo el robo de objetos en el juego, al pasarlos a tu propia cuenta desde la de otro jugador. Para evitar este tipo de trampas, se debe implementar un sistema que detecte el origen del envío de un paquete, verificando así si el envío del paquete con datos fraudulentos se está enviando desde una ubicación normal para el jugador (que no se esté enviando desde otro país al que reside el usuario, por ejemplo).

Una vez vistos los tipos de trampas existentes, se van a explicar los distintos sistemas para evitar las trampas (también llamadas *hacks*) que los desarrolladores de videojuegos suelen implementar para evitar que un jugador pueda tener grandes ventajas frente a otros usuarios, lo cual podría suponer un gran fracaso al empeorar gravemente la experiencia de juego. Estos sistemas se pueden dividir en dos categorías, dependiendo de en que parte del videojuego en red estén implementados, si en la parte cliente o en la parte servidor.

3.3.1. Métodos antitrampas implementados en el lado del servidor

Estos métodos están implementados en el servidor del videojuego y se encargan principalmente de analizar los paquetes de red recibidos por los jugadores y de comprobar que los datos y el estado del juego de este es correcto. Existen varios tipos:

- **No confiar en el jugador**: este método se basa en la verificación del servidor de todos los datos recibidos por los jugadores, comprobando si las acciones recibidas son coherentes con el estado actual del juego y la situación actual del jugador. El uso de este método es clave en cualquier videojuego en red para evitar inconsistencias en las partidas, ya que se podrán detectar paquetes de red que incluyan acciones imposibles para el usuario (sólo posibles mediante el uso de trampas).

  Un ejemplo podría ser el caso en el que, en un juego de lucha, el servidor recibe un paquete que contiene un ataque específico hacia el otro jugador. El servidor debería comprobar que ese ataque lo tiene disponible el personaje en cuestión y, en caso de que no, denegar la acción y penalizar al usuario, debido a que está intentando hacer trampas (podría ser que ese ataque es muy poderoso y que, a pesar de que su personaje no lo tiene, ha intentado ejecutarlo mediante un programa externo que envía el paquete de red modificado).

  Hay que tener en cuenta que se necesita una potencia mínima en el hardware de los servidores para realizar estas verificaciones sin que provoque interrupciones en el videojuego. Conlleva también una gran carga de trabajo para los desarrolladores, ya que deberán programar verificaciones para todas las acciones posibles en el videojuego.

- **Diseño de un protocolo de aplicación que detecte las alteraciones**: el protocolo de aplicación es el encargado de encapsular los paquetes de red y de decidir cómo se manipulan los datos contenidos en los paquetes. Para diseñar un sistema

antitrampas relacionado con esta capa, el desarrollador debe implementar verificaciones de los paquetes en la parte servidor del juego, las cuales sean capaces de detectar paquetes derivados de trampas.
El diseño del protocolo de aplicaciones dependerá del protocolo de transporte escogido en el videojuego en red, los cuales se han explicado previamente (TCP o UDP). En el caso de escoger TCP, este protocolo ya podrá evitar en gran medida paquetes fraudulentos debido al uso de los números de secuencia en cada paquete. Por ejemplo, en un intento de ataque de reproducción, el servidor recibirá muchos paquetes con un mismo número de secuencia igual, por lo que los desechará al no cumplir con el protocolo TCP.

Los paquetes mediante TCP pueden conllevar sobrecarga y grandes latencias, ya que no soportan la pérdida de paquetes. Por ello, se utiliza más el protocolo de transporte UDP en los videojuegos en red, por lo que el desarrollador deberá implementar correctamente el protocolo de aplicación y la estructura del paquete para evitar las trampas de los jugadores.

Fig. 22. Estructura común de un paquete UDP en los videojuegos en red [15]

Como se ve en la **Figura 22**, el paquete UDP típico en los vidoejuegos en red está compuesto por 4 partes, las cuales van a ser explicadas de izquierda a derecha:

- La parte con datos relacionados con el protcolo TCP, en la que se incluyen datos que suele incluir el protocolo TCP, cómo los números de secuencia para poder comprobarlos en la parte servidor. Esta implementación se suele llamar TCP sobre UDP, ya que se utilizan paquetes UDP que incluyen alguna característica del protcolo TCP (suele ser los números de secuencia, como ya se ha comentado).

- La parte en la que se indica la latencia del usuario, para tenerlo en cuenta a la hora de cómo tienen que actuar las técnicas de compensación de latencia implementadas en el videojuego en red.

- La parte de datos relacionados con el protocolo, en la que el desarrollador puede incluir datos que sirvan para verificar el paquete de forma más segura, cómo mediante el uso de hashes.

- La parte con los datos del juego, en la que se indica la acción que ha realizado el usuario y la que el servidor tendrá que tener en cuenta para el siguiente estado del juego (en caso de que el paquete se verifique cómo correcto).

El diseño correcto la capa de aplicación por parte de los desarrolladores de un videojuego en red es clave para evitar que paquetes fraudulentos relacionados con programas externos diseñados para hacer trampas puedan llegar a ser recibidos por el servidor como paquetes correctos, lo cual conllevaría que influyeran negatiavamente en el resto de usuarios del juego.

- **Ocultación del tráfico de red**: esta técnica antitrampas se basa en la encriptación de los paquetes de red enviados entre el servidor y los clientes.
  Un paquete de red sin encriptar puede ser capturado y visualizado al completo mediante programas de análisis de paquetes como Wireshark, lo cual puede servir para hacer trampas. Por ejemplo, mediante la captación de los paquetes de red, se podría saber la posición exacta de los usuarios, aunque el juego no te lo muestre, lo cual daría ventaja respecto al resto de jugadores.

  Encriptando los paquetes de red del videojuego se evita que se puedan filtrar los datos de las comunicaciones. Este proceso se lleva a cabo mediante un algoritmo de encriptación en el emisor del paquete, debiendo ser desencriptado en el receptor del paquete, utilizando para ello claves que deben ser privadas y ocultas para los jugadores. Por tanto, los desarrolladores deben implementar el algoritmo de encriptación en el videojuego en red, dificultando de esta forma la posibilidad de hacer trampas.

  También es una buena opción cambiar el algoritmo de encriptación frecuentemente mediante actualizaciones, para evitar que jugadores que hayan descubierto el algoritmo de encriptación de los paquetes, puedan hacer trampas.

- **Uso de métodos estadísticos**: los métodos estadísticos antitrampas recogen la mayor cantidad de datos posibles sobre las partidas de todos los jugadores (victorias, derrotas, muertes, etc.…) para reconocer patrones extraños que puedan significar el uso de trampas.

  Estos programas se implementan en la parte servidor del videojuego y suelen necesitar de la intervención humana para las revisiones, es decir, para definir si un patrón detectado cómo inusual está vinculado al uso de trampas o no (penalizando al jugador en caso de que sí). Esto es así debido a que la penalización automática a un jugador cada vez que el programa antitrampas detecte un patrón de juego inusual, puede llevar a un gran número de falsos positivos, lo cual arruinaría notablemente la experiencia de juego de los usuarios.

  Las revisiones humanas pueden ser realizados por trabajadores de la compañía desarrolladora del videojuego o por los propios usuarios (útil para compañías pequeñas o para videojuegos con un gran número de jugadores). En el segundo caso, cuando se detecta un patrón inusual de un jugador en una partida, otros jugadores de esa partida recibirán la pregunta sobre si ese jugador ha hecho trampas realmente. En caso de que un gran porcentaje de los jugadores digan que sí ha realizado trampas, el jugador en cuestión puede ser penalizado automáticamente o revisado más exhaustivamente por un trabajador de la compañía (de esta forma se libera la carga de revisiones).

  Este tipo de métodos antitrampas son muy utilizados debido a su efectividad y a la baja carga que supone para el servidor, ya que el programa puede ser implementado en otra máquina distinta a la del servidor del juego. Aunque hay que tener en cuenta que se necesitarán muchos datos para que sean muy efectivos, lo cual en juegos con pocos jugadores puede ser un problema y puede llevar a un excesivo número de revisiones sobre patrones de juego inusuales.

3.3.2. *Métodos antitrampas implementados en el lado del cliente*

Este tipo de métodos se implementan en los dispositivos de los jugadores (en la parte cliente), instalándose con el videojuego. Estos programas antitrampas funcionan mediante la realización de acciones en la parte cliente del videojuego y mediante el envío de datos desde la parte cliente instalada en los dispositivos de los jugadores al servidor del videojuego. Existen diversos tipos, los cuales van a ser desarrollados a continuación:

- **Encriptación del código del videojuego**: la encriptación de la sección de código de un videojuego en red en la parte cliente es algo esencial para prevenir trampas. Este método se puede llevar a cabo de diversas formas:

  - Empaquetado: se encripta todo el código del juego y se desencripta por completo al iniciar el juego mediante un método de desempaquetado específico. Es fácil de implementar, pero tiene un gran riesgo, ya que si un usuario descubre el método de desempaquetado que utiliza el juego, podrá desencriptar el código y visualizarlo para sacar ventajas.

  - Encriptación parcial: para que el proceso sea más seguro, se puede optar por encriptar todo el código y durante la ejecución del videojuego desencriptar sólo la función que se vaya a utilizar, encriptándola de nuevo después. De esta forma se evita que el juego esté desencriptado al completo mientras se está ejecutando.

  - Debido a que tanta encriptación y desencriptación podría llevar a una sobrecarga computacional y, por tanto, a la disminución del rendimiento del juego, se puede optar por no encriptar ciertas funciones durante la ejecución del juego (funciones que se utilicen recurrentemente, que no sean importantes o que conlleve mucha carga computacional encriptarlas y desencriptarlas debido a su tamaño), aumentando de esta forma el rendimiento y mejorando la experiencia de juego de los usuarios.

  La encriptación es un método antitrampas esencial que se utiliza en todos los videojuegos en red, utilizándose también para encriptar otros métodos antitrampas implementados en la parte cliente del juego, lo cual añade una capa más de protección.

- **Verificación de archivos mediante *hashing***: este tipo de método antitrampas se basa en la asignación de un código hash único a cada archivo de la parte cliente del videojuego en red. De esta forma, se puede verificar si un archivo del juego ha sido modificado en la parte cliente para hacer trampas, ya que al modificarlo su código hash cambiará.

  Un jugador puede querer modificar archivos para tener ventajas, como modificar texturas o la iluminación del juego para poder ver a los usuarios más fácilmente.

  El proceso se basa en el envío de los códigos hash desde la parte cliente al servidor. Este contiene una base de datos con todos los códigos y, por tanto, podrá verificar si el código hash ha variado, en cuyo caso penalizará al jugador por hacer trampas al modificar los archivos del juego.

Este método antitrampas es sencillo de implementar para los desarrolladores de los videojuegos, pero tiene el riesgo de la facilidad de manipulación, ya sea mediante la modificación del algoritmo de generación de códigos hash o mediante la modificación de los paquetes de red que se envían del cliente al servidor con el número de hash para la verificación. Esto se debe contemplar y evitar mediante métodos cómo la encriptación en la parte cliente del algoritmo de generación de códigos hash y mediante el método de ocultación del tráfico de red (visto previamente en los métodos antitrampas de la parte servidor del juego).

- **Detección de programas utilizados para hacer trampas**: este método antitrampas se basa en la detección en los dispositivos de los jugadores de la ejecución de software para hacer trampas. Para ello, se implementa en la parte cliente un método que escanea el dispositivo del usuario y envía al servidor datos sobre los procesos en ejecución, incluyendo su identificador hash, el nombre de ventana y el nombre del proceso. Una vez llegan los datos al servidor, este los contrasta con una base de datos que incluye herramientas utilizadas para hacer trampas, pudiendo de esta forma identificar si alguno de los procesos de dispositivo del jugador coincide con alguna de ellas (penalizando al jugador en caso afirmativo).

  Sin embargo, este método antitrampas es sencillo de manipular para los *hackers*, ya que pueden optar por modificar el escáner para que obtenga datos falsos o incluso modificar los paquetes de red enviados al servidor en los que se envía la información de los procesos ejecutados en el dispositivo. Para evitar esto, es necesario utilizar otros métodos antitrampas complementarios, cómo la encriptación del código del juego (encriptando la parte del código que escanea el dispositivo), la ocultación del tráfico de red (para evitar la modificación de los paquetes) y la ocultación de la memoria (método que se explicará a continuación).

  Debido a que este método no interfiere con la lógica del juego, no afectará al rendimiento de este, ya que el escaneo del dispositivo del jugador se realiza en paralelo a la ejecución del videojuego, por lo que es un método antitrampas eficaz y óptimo para los desarrolladores si se utiliza junto a otros métodos.

- **Ocultación de la memoria**: este método antitrampas está basado en la encriptación e incluso reasignación de la posición de memoria de las variables contenidas en la pila de memoria de un dispositivo, la cual es utilizada por el código del juego y por otros métodos antitrampas.

  Los *hackers* pueden acceder a la pila de la memoria de su dispositivo mediante programas para hacer trampas y así visualizar los cambios de los valores de las variables. Con ello, pueden acabar descubriendo como es el funcionamiento del código del juego o de otros métodos antitrampas que también usan la pila de la memoria, además de conocer valores ocultos para los usuarios, como posiciones o ciertas características de un personaje.

  Para evitar esto, se debe encriptar el valor de las variables e incluso proceder a la reasignación de la posición de memoria cuando el valor de las variables cambie, ya que de esta forma su valor estaría protegido aún incluso si el *hacker* consiguiese averiguar el algoritmo de encriptación de las variables.

Sin embargo, la reasignación puede llevar a una alta sobrecarga computacional que afecte al rendimiento y la experiencia de juego de los usuarios, por lo que la mejor opción es únicamente aplicar la reasignación a las variables más críticas, de las cuales el usuario no debe conocer el valor. Para el resto, se les puede aplicar únicamente la encriptación o incluso no aplicarles nada en caso de que se traten de valores conocidos para los jugadores, como las coordenadas de un vehículo.

La ocultación de la memoria es un método muy útil para proteger el código del juego al ejecutarse, así como a otros métodos antitrampas que utilicen la memoria. Aunque es recomendable utilizarlo junto al método de encriptación del código del juego, para tener una capa más de protección y ponérselo aún más difícil a los *hackers*.

- **Controlador antitrampas basado en el kernel**: este método antitrampas está implementado a nivel de kernel, por lo que tiene acceso total a la memoria, al hardware, a las instrucciones de la CPU y a las estructuras de datos del sistema operativo. De esta forma, puede detectar, por ejemplo, los intentos de manipular la memoria o los datos de un juego en ejecución mediante un software para hacer trampas, pudiendo bloquear el intento y penalizando al jugador.

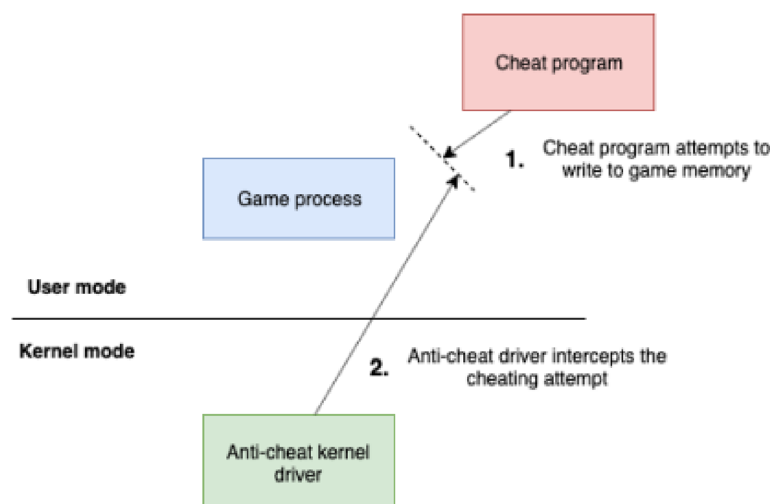


Fig. 23. Representación del funcionamiento del controlador antitrampas basado en el kernel [15]

En la **Figura 23** se puede observar como funcionan estos controladores mediante la división entre el modo usuario y el modo kernel. En la imagen se ve un intento de un usuario de hacer trampas en un videojuego mediante un programa externo para hacer trampas. Cuando el programa externo intenta modificar la memoria del juego, el controlador antitrampas instalado en el kernel lo detecta e impide que la modificación se lleve a cabo.

Este tipo de programas antitrampas son muy eficaces y muy difícil de manipular, ya que operan a muy bajo nivel y no están asociados al ejecutable del videojuego en red. Para sobrepasar este método, un *hacker* necesitaría desarrollar un software que opere a nivel de kernel y en el caso de Windows y MacOS el software debe estar firmado para operar a nivel de kernel, para lo cual los propios Microsoft y Apple deben darle el certificado (lo cual no obtendrá si no es un software de

confianza). Aun así, los *hackers* podrían sobrepasar el inconveniente de la firma si el videojuego en red tiene alguna vulnerabilidad en el controlador antitrampas que opera en el kernel, ya que podría manipularlo.

El método antitrampas basado en el kernel es el más efectivo de los utilizados en la parte cliente, ya que es muy complicado de manipular si está bien implementado. Sin embargo, es complejo de desarrollar, ya que hay que tener amplios conocimientos sobre como opera el sistema operativo con los procesos y sobre cómo funciona el kernel, siendo este la interfaz que comunica el hardware y los procesos en un ordenador. Por lo que se trata de un tipo de programación muy complejo, lo cual implica que solo se deba implementar este método antitrampas cuando se tengan grandes conocimientos sobre el tema, para evitar que puedan existir vulnerabilidades que puedan aprovechar los *hackers*.

También es recomendable utilizar este método junto a otros para aumentar la protección contra las trampas, pudiendo, por ejemplo, incluir otros métodos antitrampas en el controlador desarrollado a nivel de kernel, lo cual haría que esos métodos sean más complejos de ser manipulados por los *hackers* al tener varias capas de protección.

4. Diseño de los prototipos de videojuegos en red

Una vez acabada la parte teórica, se van a aplicar los conocimientos explicados mediante el desarrollo de dos prototipos de videojuegos en red en el motor de videojuegos Unity. Un motor de videojuegos es un software que permite crear de forma completa un videojuego. Esto es posible gracias a la diversidad de funciones que ofrece [33]:

- Un motor gráfico 2D y 3D
- Un motor físico para simular magnitudes como la aceleración, velocidad, fuerza, gravedad, etc.
- Funcionalidad para crear animaciones.
- Funcionalidad para añadir sonidos al juego.
- Posibilidad de implementar inteligencias artificiales.
- Programación de la parte lógica mediante el lenguaje C# (en el caso de Unity).
- Herramientas para desarrollar la parte red de los videojuegos.

Unity es uno de los principales motores de videojuegos utilizados en la industria junto a Unreal Engine. No existe un motor de videojuegos mejor que el otro, si no que dependiendo del proyecto que se vaya a realizar habrá que realizar un estudio sobre cual es más conveniente. En mi caso, que nunca he utilizado un motor de videojuegos previamente, la opción que mejor se adapta es el uso del motor Unity debido a que tiene una comunidad más grande y existe más material de aprendizaje sobre este en Internet, lo cual ayuda considerablemente en la fase de formación en el uso del motor. Unity es un motor ideal para comenzar en el sector del desarrollo de videojuegos, mientras que Unreal Engine ofrece como principal ventaja la posibilidad de crear videojuegos con los gráficos más realistas, usándose por tanto en la mayoría de los videojuegos AAA [34].

El objetivo es que haya diferencias notables entre los dos prototipos para así mostrar juegos con géneros, arquitecturas de red, niveles de complejidad y técnicas y procesos de red diferentes. Para ello, primeramente, hay que llevar a cabo la fase de diseño de los prototipos, la cual se va a tratar en este apartado del documento. En esta fase se van a definir los siguientes conceptos para cada prototipo de videojuego en red:

- Género del prototipo.
- Arquitectura de red a utilizar.
- Nivel de complejidad al que pertenece.
- Técnicas de compensación de latencia que se van a implementar.
- Como va a ser el sistema de *matchmaking*.
- Gestión de las trampas en el prototipo.
- Funcionamiento esperado.
- Herramientas que utilizar para implementar la parte red.

4.1. Diseño de prototipo de shooter multijugador en tercera persona

El primer prototipo de videojuego en red va a consistir en un shooter en tercera persona (TPS) multijugador. Un juego de este género requiere de latencias lo más bajas posibles con tiempos de respuesta mínimos y una sincronización precisa entre los estados de juego de los jugadores en la partida. Si no se cumplen estos requisitos en juego de este tipo, se pueden generar inconsistencias que arruinen la experiencia de juego de los usuarios.

Respecto a la arquitectura de red a utilizar, se va a optar por implementar la de cliente-servidor con servidor dedicado. Se considera que esta opción es la más correcta para este género debido a factores como el equilibrio de latencias entre los jugadores al no haber un *host* que tenga ventaja al tener latencia nula, lo cual aporta una competitividad más justa. Además, se puede controlar que la conexión a internet del servidor sea la mejor posible (lo cual con un *host* no depende de los desarrolladores si no del propio jugador), aportando así una mayor consistencia y estabilidad a las partidas al ser el encargado de sincronizar los estados de juego de los jugadores.

En comparación con la arquitectura *peer-to-peer*, mejora notablemente los tiempos de respuesta, latencias y sincronización de los estados de los jugadores debido al incremento notable en la velocidad de conexión y en el rendimiento de cada jugador, ya que se evita que cada usuario tenga que realizar una conexión con cada uno de los jugadores de la partida, lo cual puede llegar a consumir una gran cantidad de recursos.

Esta arquitectura de red aporta también una mayor seguridad al videojuego debido a que evita la exposición de las direcciones IP entre los distintos jugadores que haya en una partida al no haber conexión directa entre ellos y sólo teniendo que realizar una conexión al servidor de la partida. Esta arquitectura es también la mejor frente a las trampas de los jugadores, ya que es el servidor dedicado el encargado de aplicar los cambios relevantes en el mundo y el estado de juego de los jugadores. Otro motivo para utilizar esta arquitectura es la mejora notable en la disponibilidad, ya que esta depende directamente de los desarrolladores del videojuego, mientras que con un *host* depende de que el jugador

que hace la función no salga de la partida, en cuyo caso la partida acabaría o habría que realizar el tedioso proceso de una migración de *host*.

Teniendo en cuenta el género elegido para este prototipo, se puede clasificar a este juego en el nivel 3 en referencia a su complejidad de la parte red. Este nivel, como se ha explicado previamente, corresponde a los videojuegos multijugador en tiempo real, donde todos los jugadores de la partida interactúan entre ellos de forma simultánea y constante, siendo de esta forma la arquitectura cliente-servidor con servidor dedicado la más adecuada para este nivel de complejidad de red.

La latencia es uno de los factores más importantes de este género de videojuegos en red si no el que más, por lo que es esencial implementar técnicas que la compensen para que la experiencia de juego sea lo más fluida posible para los jugadores.

Tras hacer una investigación sobre las posibilidades de aplicar técnicas de compensación de latencia en un videojuego en red con Unity, se llega a la conclusión de que las más factibles y útiles de utilizar son las técnicas basadas en la predicción. Estas técnicas son muy efectivas en reducir la latencia y son las más utilizadas en la industria de los videojuegos, mejorando la fluidez mediante la predicción de estados de juego futuros o pasados. Más adelante, dependiendo de la herramienta que se elija para implementar la parte red del prototipo, se verá que técnicas dentro de la rama de la predicción se utilizan exactamente.

Respecto al sistema de *matchmaking* que se va a utilizar en este prototipo, se pretenden crear sesiones de juego con hasta un máximo de 5 jugadores por sesión (cada una de las cuales será creada y gestionada por un servidor dedicado). Cada sesión dispondrá de un mapa, teniendo la opción de elegir entre mapas distintos. Al jugador se le dará la opción de elegir el mapa en el que quiere jugar la partida, por lo que el sistema de *matchmaking* se encargará de unirle a una sesión de juego con el mapa elegido y la cual tenga menos de 5 jugadores conectados.

Después de deliberar sobre los distintos métodos antitrampas disponibles y explicado previamente en este documento, se llega a la conclusión de que, con las opciones que existen en Unity, la opción más factible en este género es implementar de manera parcial el método antitrampas de no confiar en el jugador. Se busca la implementación parcial debido a que no se desea el exceso de comprobaciones por parte del servidor, lo cual podría suponer un aumento en los tiempos de respuesta, lo cual es uno de los factores claves en este género. Sin embargo, al aplicar la arquitectura de red de servidor dedicado, se quiere mantener la autoridad del servidor, llevando a que las modificaciones críticas en el mundo del juego se realicen en el lado del servidor.

Es decir, en este prototipo se decide priorizar la fluidez en la jugabilidad y la minimización de los tiempos de respuesta y la latencia. No se contempla el uso del resto de técnicas antitrampas explicadas previamente debido a que algunas son complejas de integrar con Unity y otras suponen una gran carga de recursos para el cliente o el servidor, lo cual influiría negativamente en los factores que se priorizan en este prototipo.

El siguiente concepto que tratar, es el funcionamiento que se desea en este prototipo de videojuego en red. Debido a que este trabajo de fin de grado se centra en la parte red de los videojuegos, lo relevante será la correcta implementación de esta parte para

comprender en detalle y poner en práctica los conceptos investigados y aprendidos en la parte teórica del trabajo. Es decir, es importante tener en cuenta que este trabajo no se centra en el desarrollo y creación de videojuegos, si no en el desarrollo de la parte red de estos.

Teniendo en cuenta lo comentado, la intención es simplificar el funcionamiento del prototipo y centrarse en la parte red. La idea principal es realizar un shooter multijugador en tercera persona con Unity 3D en el que cada uno de los mapas este compuesto por un plano con obstáculos representados por bloques. Cada uno de los usuarios dispondrá de un personaje que será generado aleatoriamente en cualquier punto del mapa al entrar en una sesión de juego.

Cada personaje poseerá una *skin* que será una figura geométrica simple, la cual portará un arma. El personaje tendrá las opciones de moverse en horizontal y en vertical mediante las teclas AWSD o mediante las flechas del teclado y también podrá saltar pulsando la tecla de espacio. Respecto a la funcionalidad de disparo del arma, se podrá disparar pulsando el botón izquierdo del ratón y se podrá recargar el arma de balas con el botón derecho. Cada personaje tendrá una cierta cantidad de salud que se irá restando cuando le impacten las balas de los oponentes. Cuando el personaje llegue a 0 o menos salud o bien se salga de los límites del mapa, este morirá y se volverá a regenerar en un punto aleatorio del mapa.

La duración de la partida será infinita o hasta que el servidor dedicado se desconecte. Es decir, cualquier jugador puede entrar y salir en cualquier momento y no hay límite de muertes, por lo que se podría decir que cada sesión de juego del prototipo consiste en una partida de entrenamiento en la que el único objetivo de cada usuario es aprender y mejorar sus habilidades.

Una vez descritos todas las características que se desean en este prototipo, la última parte de la fase de diseño será la elección de la herramienta que se va a utilizar para implementar la parte red del videojuego. Los principales frameworks de red que se integran con Unity son FishNet, Mirror, Photon y la propia herramienta de Unity, llamada Netcode for GameObjects. Estos frameworks facilitan enormemente la programación de la parte red en los videojuegos al proporcionar scripts con clases y métodos ya creados, los cuales ahorran mucho trabajo a los desarrolladores al proporcionar diversas funciones con simplemente integrar el framework en el proyecto de Unity.

Tras realizar una investigación para conocer detalladamente las funcionalidades que aporta cada una de las principales herramientas para implementar la parte red del prototipo, se llega a la conclusión de que la que más se adapta es la de Photon, en concreto la de Photon Fusion, ya que existen diversos frameworks creados por Photon.

Photon Fusion es la más adecuada para este prototipo debido a que es la que más servicios ofrece y más facilidades da para implementar la parte red. Los principales motivos para utilizar este framework de red son las ventajas que ofrece respecto a la banda ancha (proporcionando una gran velocidad de red), la gran consistencia en la transferencia de datos y la integración directa con esta herramienta de técnicas de compensación de latencia como la interpolación (técnica basada en la predicción de estados de juego anteriores para aumentar la fluidez) y la asistencia de control (técnica basada en el ajuste del mundo del juego para compensar acciones en las que un jugador pueda ser perjudicado

por una mayor latencia) [18]. Además de todo lo mencionado, Photon Fusion también dispone de una API para implementar el *matchmaking* del prototipo [21]. Estos factores son de importancia clave para este prototipo, por lo que poder integrarlos de forma sencilla es una gran ventaja. La forma de implementarlos se explicará más adelante en el documento en el apartado de implementación.

TABLA I
Diferencias entre las características de Photon PUN, Bolt y Fusion

	PUN 2011	Bolt 2014	Fusion 2021
Target Player Count	32	32-50	200
Core-Features			
Tick based simulation	✗	✓	✓
Client Side Prediction	✗	✓	✓
Lag Compensation	✗	✓	✓
Snapshot Interpolation	✗	✓	✓
Replication System			
Delta Snapshots	✗	✗	✓
Eventual Consitency	✗	✓	✓
Performance			
Allocations (Runtime)	/	/	zero
Performance (Benchmarks)	+	+	+++
Bandwidth	/	+	+++
Bespoke Prebuilt Functionality			
Area of Interest (AOI)	✗	✗	✓
Network Animator	✗	✗	✓
Network Character Controller (KCC)	✗	✗	✓
Auth. Rigidbodies w/ CSP	✗	✗	✓

Nota: se muestran las diferencias entre las distintas herramientas de red de Photon en lo referente a rendimiento, características integradas, replicación y funcionalidades preconstruidas [18]

En la **Tabla 1** se pueden observar las principales diferencias entre los distintos frameworks que ofrece Photon, pudiendo ver las ventajas que se han comentado. Estas características no se encuentran integradas directamente tampoco en el resto de frameworks mencionados, lo cual pone a Photon Fusion un escalón por encima en videojuegos en los que se requiera de bajas latencias para que la experiencia de juego sea correcta.

Este framework de red, como la mayoría, utiliza el protocolo de transporte UDP, el cual es la mejor opción para el sector de los videojuegos debido a su mayor velocidad de transmisión de datos en comparación con el protocolo TCP. Esto es aún más importante en un videojuego en red de género shooter al necesitar tiempos de respuesta mínimos.

4.2. Diseño de prototipo de juego de cartas multijugador basado en turnos

El segundo prototipo de videojuego en red que se va a realizar consiste en un juego de cartas multijugador basado en turnos. Este género de videojuegos tiene grandes diferencias con respecto al género shooter, ya que los tiempos de respuesta y latencia no serán tan importantes y latencias incluso de unos segundos no afectan apenas a la calidad de experiencia del juego. En este género lo más importante es la consistencia de los datos y evitar la manipulación o pérdida de estos, ya que supondría un gran impacto en la partida.

La arquitectura de red que se va a utilizar va a ser también la cliente-servidor con servidor dedicado. Sin embargo, en el caso de este prototipo, no se elige ese tipo de arquitectura por las ventajas que ofrece en la latencia y los tiempos de respuesta (ya que en este género no es importante minimizar estas características), si no por la gran seguridad que puede ofrecer frente a las trampas, las cuales son un factor clave en este género.

Por tanto, aprovechando las posibilidades que ofrece esta arquitectura, se va a aplicar el método antitrampas de no confiar en el jugador de forma completa. Es decir, no solo se va a requerir que las modificaciones importantes en la partida se hagan en el lado del servidor, sino que también se van a realizar diversas comprobaciones de los datos que envían los jugadores en la ejecución de cada turno de la partida para asegurar que no se llevan a cabo trampas que alteren considerablemente la experiencia de juego.

Este género de videojuegos se clasifica en el nivel 2 en lo referente a su complejidad de la parte red, ya que está basado en turnos. Este nivel, como ya se ha comentado previamente, no requiere de baja latencia o bajos tiempos de respuesta, ya que, a diferencia del primer prototipo, en ningún momento de la sesión de juego los dos usuarios estarán interactuando a la vez, siendo de esta forma una actividad de red asíncrona.

Debido a que la latencia no es importante en este prototipo, se ha decidido prescindir del uso de técnicas de compensación de latencia, centrando el objetivo en la seguridad como ya se ha comentado.

En este prototipo el sistema de *matchmaking* que se va a implementar va a ser más simple, optando por la creación de sesiones de juego en las que habrá un máximo de 2 jugadores (creadas y gestionadas por un servidor dedicado como en el primer prototipo). En este caso no existe el filtrado al unirse a una partida por lo que, cuando un usuario inicie el juego, el sistema únicamente buscará una sesión en la que haya menos de 2 jugadores, intentando llenar las máximas posibles. Si el jugador entra a una sesión vacía, esperará en esta hasta que se un oponente se una. Se plantean también otros requisitos para el sistema de *matchmaking* en este prototipo para mejorar la experiencia de juego:

- Si durante el transcurso de una partida uno de los jugadores abandona la sesión (ya sea por voluntad propia o por desconexión), el otro jugador será notificado de lo ocurrido y se le mantendrá a la espera de otro jugador en esa misma sesión para iniciar una nueva partida en la que se reestablecerá el estado inicial del tablero.

- En caso de desconexión del servidor durante una partida, ambos jugadores serán notificados y se les cerrará automáticamente el juego para que puedan volver a iniciarlo en busca de una nueva partida.

Respecto al funcionamiento que se desea en este prototipo de videojuego en red, se va a implementar un juego de cartas multijugador basado en turnos y en partidas de dos jugadores cada una. Cada jugador dispondrá de 20 vidas y de 10 cartas, cada una de las cuales corresponderá a un personaje que tendrá un determinado valor de ataque y defensa. Cada jugador podrá tener un máximo de 5 cartas sobre el tablero, de las cuales podrá elegir una de ellas para atacar en su turno. El jugador oponente será informado de la carta con la que se le está atacando y elegirá una de sus cartas para defender. Para finalizar el turno, se le restarán al jugador defensor las vidas correspondientes al valor de la resta entre el ataque de la carta atacante y la defensa de la carta defensa (si la defensa es mayor que el ataque no se le restará ninguna vida). Los dos jugadores podrán coger cartas de su montón en cualquier momento de la partida en caso de que tengan menos de 5 sobre el tablero. Después de usar una carta, esta se irá al montón de las descartadas, el cual se podrá mezclar en cualquier momento con el montón en el que se encuentren las cartas que aún no se han puesto sobre el tablero. La partida acabará cuando uno de los dos jugadores llegue a 0 o menos vidas, en cuyo caso habrá perdido.

Debido a que se trata de un juego de cartas en el que no es necesario el uso de tres dimensiones, se va a implementar este prototipo mediante Unity 2D, implementando de esta forma un prototipo en 3D y otro en 2D.

También se va a optar por utilizar una herramienta distinta al primer prototipo para implementar la parte red, pudiendo así ver las diferencias de implementación entre distintas herramientas. Se va a utilizar la herramienta creada por Unity, denominada como Netcode for GameObjects. De esta forma se podrá concluir si merece la pena el uso de frameworks de red de terceros como Photon Fusion o es más conveniente utilizar la herramienta proporcionada por Unity. Photon Fusion es la herramienta de red más completa en cuanto a las características que ofrece, pero Netcode for GameObejcts puede llegar a ser más sencilla de integrar con Unity al haberla desarrollado. En este prototipo, debido a que no se va a enfocar en la latencia y en la velocidad de transmisión, no se aprovecharían las funciones extra que ofrece Photon Fusion (alta velocidad, técnicas de compensación de latencia integradas directamente…), por lo que es una buena opción optar por el uso de la herramienta de red desarrollada por Unity.

Respecto a la implementación del *matchmaking* Unity ofrece los Unity Gaming Services, los cuales consisten en servicios alojados en la nube con los que dar un gran salto de calidad en el desarrollo de un videojuego y facilitando la implementación de funcionalidades que de otra forma serían muy tediosas. De todos los servicios que se ofrecen, para este caso, los más convenientes a utilizar son el servicio de *Relay* y el de *Lobby*. El servicio de *Relay* será el encargado de conectar a los jugadores, mientras que el servicio de *Lobby* agrupará a los jugadores en sesiones de juego [31][32]. Es decir, cada servidor dedicado se va a encargar de crear un *lobby* (sala o sesión de juego) al que los jugadores se unirán haciendo uso a su vez del servicio de *Relay* para establecer la conexión con el servidor. El uso de estas dos herramientas se detallará en el siguiente apartado, en el que se va a desarrollar la fase de implementación de los prototipos.

Para finalizar con el apartado de diseño de los prototipos, en la **Tabla 2** se muestra una tabla en la que se podrán observar las características de cada prototipo, pudiendo apreciar así sus notables diferencias.

TABLA II
Comparación de las características de red de los dos prototipos a implementar

Características de red	Prototipo 1	Prototipo 2
Género	Shooter en tercera persona	Cartas basado en turnos
Arquitectura de red	Servidor dedicado	Servidor dedicado
Nivel de complejidad	Nivel 3: multijugador en tiempo real	Nivel 2: multijugador basado en turnos
Técnicas de compensación de latencia	Interpolación y asistencia de control	Ninguna
Matchmaking	Sesiones de juego de 5 jugadores con filtrado por mapa	Sesiones de juego de 2 jugadores incluyendo la gestión de desconexiones
Gestión de las trampas	Aplicación parcial del método no confiar en el jugador	Aplicación completa del método no confiar en el jugador
Framework para implementar la parte red	Photon Fusion	Netcode for GameObjects

5. Implementación de los prototipos de videojuegos en red en Unity

Una vez se ha establecido como va a ser y con que herramientas se va a implementar cada uno de los prototipos, se procede a la fase de implementación, en la que se va a describir detalladamente como se ha implementado cada uno de los juegos, así como las formaciones previas que se han hecho para poder llevar a cabo el desarrollo de estos.

5.1. Implementación del prototipo de videojuego shooter multijugador en tercera persona

El primer prototipo de videojuego en red que se va a implementar es el shooter en tercera persona. Debido a que es la primera vez que utilizo un motor de videojuegos, requiero de una formación previa tanto en el motor Unity como en el framework Photon Fusion con el que se va a implementar la parte red del juego.

5.1.1. Fase de formación en las herramientas a utilizar

Por tanto, antes de explicar como se ha desarrollado este prototipo, se va a describir como se ha llevado a cabo la formación para aprender a utilizar las herramientas necesarias.

Lo primer de todo es llevar a cabo una formación introductoria al motor de videojuegos Unity, aprendiendo todas las funcionalidades que tiene, así como aprender a programar en el lenguaje C#, ya que nunca he programado en él. C# es un lenguaje de programación multiparadigma orientado a objetos y creado por Microsoft, ejecutándose en su plataforma .NET [35]. Este lenguaje deriva de lenguajes como C, C++, Java o JavaScript, con los cuales, si he trabajado a lo largo de mi grado en Ingeniería Informática, lo cual facilitará enormemente el aprendizaje del lenguaje C#.

El primer contacto con Unity se ha llevado a cabo mediante una lista de reproducción de YouTube llamada *Unity Create a Game Series* [16], en la cual se desarrolla desde cero un videojuego shooter en tercera persona para un jugador, el cual consiste en oleadas de enemigos contra los que hay que sobrevivir. Cada oleada consta de un mapa distinto compuesto por obstáculos en forma de bloques. Además, el juego está dotado de música y efectos de sonido y un menú en el que modificar opciones como la resolución o el volumen del sonido del juego.

Mediante esta lista de reproducción y su proyecto subido en GitHub [17], he sido capaz de aprender a utilizar gran parte de las funcionalidades que ofrece Unity para desarrollar videojuegos: creación y funcionamiento de los *GameObjects* (objetos de Unity), manejo de escenas, manejo de las cámaras, programación en C# orientada a Unity (clases, métodos, corutinas que funcionan mediante hilos, manejo de eventos..), uso de componentes en los *GameObjects* (script para la lógica, *collider* para la gestión de las colisiones entre objetos, *rigidbody* para la simulación de las leyes físicas, mesh o malla para la *skin* básica, material para el color, *transform* para la posición y rotación, etc.), creación de *Canvas* para desarrollar IUs (interfaces de usuario), uso de *Sprites* y *Particle Systems* para los efectos visuales, integración de música y efectos de sonido, gestión de paquetes de Unity, uso de los *Nav Mesh Agents* para programar como deben moverse los objetos no jugables (como los enemigos), creación de *prefabs* (*GameObjects* dentro de las carpetas de activos del proyecto, actuando como una plantilla de la que se pueden crear instancias)…

Con todas las funcionalidades de Unity enseñadas en esta lista de reproducción, he aprendido a desarrollar multitud de elementos que se incluyen en gran parte de los videojuegos: controlador de movimiento del personaje de un usuario, creación de *skins* básicas para objetos mediante objetos 3D (cubos, cápsulas, esferas, cilindros…), creación de mapas, programación de enemigos autónomos, sistema de *spawn,* creación de un menú de usuario, manejo de obstáculos en el mapa, gestión de un sistema daños y de salud, creación de niveles con distintas dificultades…

Debido a que la lista de reproducción para aprender a utilizar Unity se basa en un videojuego para un jugador sin parte red, el siguiente paso en la fase de formación será la investigación acerca de la implementación de la parte red en un videojuego mediante el framework de red Photon Fusion, que ha sido el elegido para este prototipo debido al gran número de características que trae integradas.

Los tutoriales con los cuales he podido aprender a utilizar Photon Fusion han sido dos listas de reproducción de YouTube, denominadas *Unity / PHOTON FUSION 101 Tutorials* [19] y *Tutorial series: Online multiplayer FPS with Photon Fusion & Unity* [22] en las que se ven todos los conceptos necesarios para poder desarrollar un videojuego en red con Photon Fusion [18]:

- Integración con Unity: se debe descargar e importar el paquete correspondiente en el editor de Unity.

- Como crear un sistema de unión a una sesión online: cada cliente o servidor consta de una instancia de un objeto integrado con Photon Fusion denominado como *NetworkRunner*. Este es el núcleo de Photon Fusion, ya que es el encargado de iniciar la ejecución de la parte red (como cliente, host o servidor) y realizar acciones como crear una sesión de juego o unirse a una sesión de juego.

- Como convertir un *GameObject* en un *NetworkObject* para que ese objeto esté presente y sincronizado en el estado de juego de todos los jugadores de la sesión (un personaje de un jugador debe ser un *NetworkObject*, por ejemplo), teniendo una ID única en la partida.

- Uso de componentes que hereden de la clase *NetworkBehaviour* (que solo puede usarse en *NetworkObjects)*, integrada con Photon Fusion. Los principales son:
    - Variables *Networked*: es una propiedad que se puede añadir a cualquier tipo primitivo y a cualquier estructura o referencia a otro *NetworkObject.* La variable debe ser atributo de una clase que herede de *NetworkBehaviour*. El valor de la variable será el mismo para ese *NetworkObject* en todos los estados de juego de los clientes y el servidor de la sesión y solo podrá ser modificado por el servidor de la sesión de juego. Se le podrá añadir un método que será ejecutado en el servidor y en todos los clientes de la sesión cada vez que el valor de la variable sea modificado. Es la forma que ofrece Photon Fusion de sincronizar un valor entre todos los participantes de la sesión.
    - *NetworkTransform*: es un componente que permite mantener sincronizado el *transform* (posición y rotación) de un objeto en red en los estados de juego de todos los clientes.
    - *NetworkRigidbody*: es un componente que permite mantener sincronizado el *rigidbody* (físicas en un objeto) de un objeto en red en los estados de juego de todos los clientes.
    - *NetworkCharacterController*: es un componente integrado por Photon Fusion que facilita la creación de un sistema de control de movimiento para cada personaje de cada jugador en la sesión de juego.

- Interfaz *INetworkRunnerCallbacks*: es una interfaz que permite implementar en una clase los métodos de devoluciones de llamadas del *NetworkRunner* [36]. Se podrán crear métodos que serán ejecutados en ocasiones como la conexión del *NetworkRunner* a un servidor, cuando ocurre un fallo de conexión, cuando un jugador se une a la sesión, cuando un jugador sale de la sesión, cuando un jugador realiza un input, cuando un jugador está buscando una sesión…

- Sistema de ciclos en la ejecución: simulación de los frames originales utilizados en Unity mediante los denominados ticks introducidos por Photon Fusion. Esto hace que en las clases el método *Start()* sea sustituido por *Spawned()* y el método *FixedUpdate()* por *FixedUpdateNetwork()* para un mejor funcionamiento en la parte red debido a la diferencia de los momentos en los que se ejecutan los métodos.

- Sistema de entradas: lo primero, es crear una estructura que herede de la interfaz de Photon Fusion *INetworkInput*. Esta estructura guardará todos los inputs que se produzcan mediante el hardware del usuario en el método *OnInput*, el cual es uno de los métodos que se implementan con la interfaz *INetworkRunnerCallbacks*, la cual se ha explicado previamente. Después, dentro del *FixedUpdateNetwork()* de alguna clase, se leerán las entradas mediante el método *GetInput()* y se podrán programar las acciones correspondientes dependiendo de la entrada recibida (pudiendo, por ejemplo, utilizar métodos del *NetworkCharacterController* para ejecutar acciones de movimiento básicas).

- RPC (*Remote Procedure Calls*): se utilizan principalmente para solicitar modificaciones ala servidor que no pueda hacer el cliente directamente debido a la autoridad exclusiva del servidor. No se va a hacer uso de las llamadas remotas en este prototipo debido a que se desean conseguir los menores tiempos de respuesta posibles.

- Gestión de trampas: el servidor o host de la sesión tendrá la autoridad exclusiva sobre todos los objetos de red. Los clientes sólo podrán realizar modificaciones en estos mediante el envío de entradas a las que el servidor reaccione o solicitando la modificación mediante RPC.

- Técnicas de compensación de latencia: como implementar las técnicas de interpolación y de asistencia de control, lo cual se verá de forma más detallada en la fase de desarrollo.

5.1.2. Fase de desarrollo del prototipo de videojuego en red

Tras llevar a cabo toda la formación necesaria, comienza la fase de desarrollo del prototipo de shooter multijugador en tercera persona. Debido a que este trabajo de fin de grado se centra en la parte red de los videojuegos, en vez de desarrollar el prototipo desde cero se va a tomar como base el juego de shooter en tercera persona para un jugador que se ha visto en el tutorial *Unity Create a Game Series* [17]. El objetivo es modificar este proyecto de tal forma que se transforme en un shooter en tercera persona multijugador online en el que en cada partida cada usuario tenga un personaje y su objetivo sea disparar al resto de personajes. Para ello, habrá que eliminar las partes que no sean relevantes, modificar otras para adaptarlas a un videojuego en red y crear de cero lo que sea necesario. Parte de la base de la parte red se extrae del tutorial *Unity / PHOTON FUSION 101 Tutorials* y el de *Tutorial series: Online multiplayer FPS with Photon Fusion & Unity*, como ya se ha comentado previamente [20] [22].

Se va a explicar detalladamente cada una de las funcionalidades de este prototipo ya modificado y adaptado para ser un videojuego en red desarrollado mediante Unity 3D y el framework Photon Fusion. Como ya se ha mencionado en la fase de diseño, es un juego con arquitectura cliente-servidor con servidor dedicado, por lo que se describirá lo que se ha implementado para cada una de las dos partes.

La primera parte que explicar va a ser la de lado del servidor dedicado. Debido a que el servidor no necesita de ninguna interfaz gráfica, se ha optado por utilizar la funcionalidad de *Dedicated Server* que ofrece Unity [37], mediante la cual se permite que el archivo ejecutable que se crea mediante el proceso de *build*, sea un ejecutable sin gráficos, tal

como se puede ver en la **Figura 24**. De esta forma, la ejecución del servidor consumirá menos recursos del dispositivo en el que se ejecute.

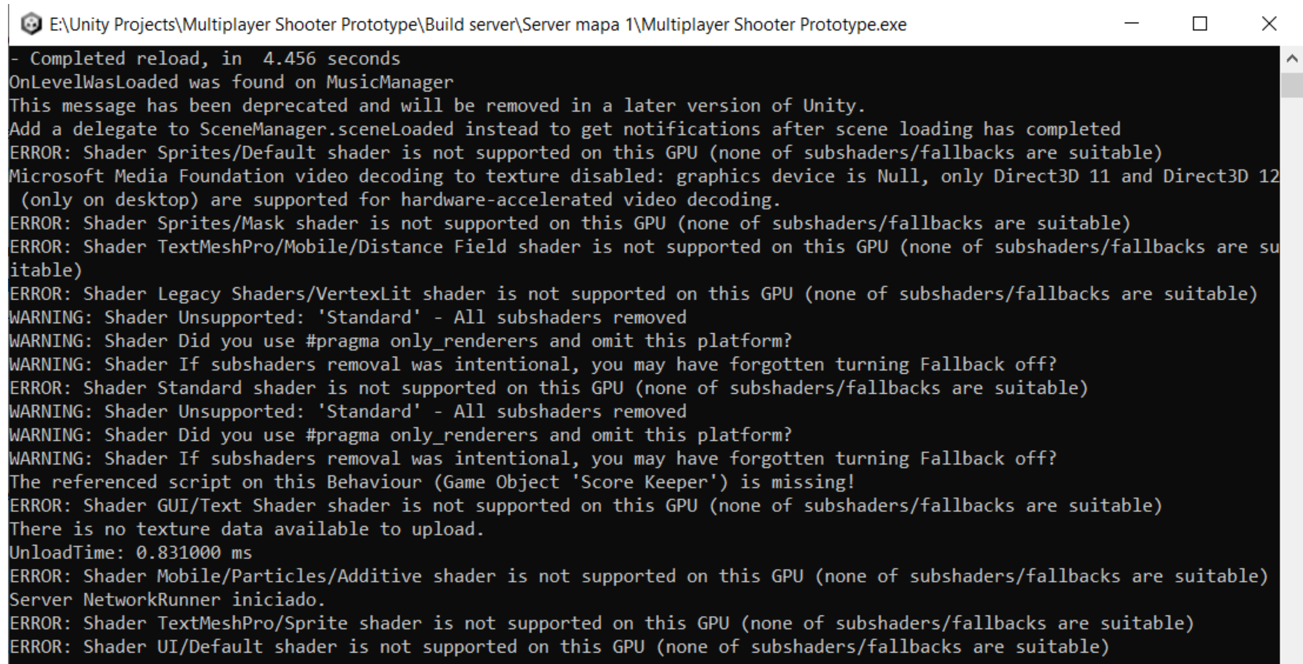


Fig. 24. Ejecución de un servidor dedicado en el prototipo de shooter multijugador en tercera persona

Todos los errores y warnings que aparecen al ejecutar un servidor son debido a todos los tipos de elementos gráficos que no se pueden ejecutar, aunque esto no afecta al funcionamiento. Estos mensajes no se pueden desactivar (incluso aun ejecutando una escena de Unity vacía aparecen), por lo que es algo que Unity debería arreglar en el futuro para que la ejecución sea más limpia sin esa sobrecarga de mensajes innecesarios al no afectar en la ejecución del servidor.

Para poder hacer uso de esta función, habrá que añadir a Unity el módulo de *Windows Dedicated Server Build* (en caso de que el server se quiera ejecutar en Windows). De esta forma, en el menú para realizar el *build* se habilitará la opción para hacerlo en modo *Dedicated Server*, tal como se ve en la **Figura 25**. Habrá por tanto dos *builds* distintos que generarán ejecutables distintos, uno para los servidores dedicado (sin parte gráfica en el modo *Dedicated Server*) y otro para los clientes/jugadores (el ejecutable con todos los gráficos creado en la sección Windows, Mac, Linux).

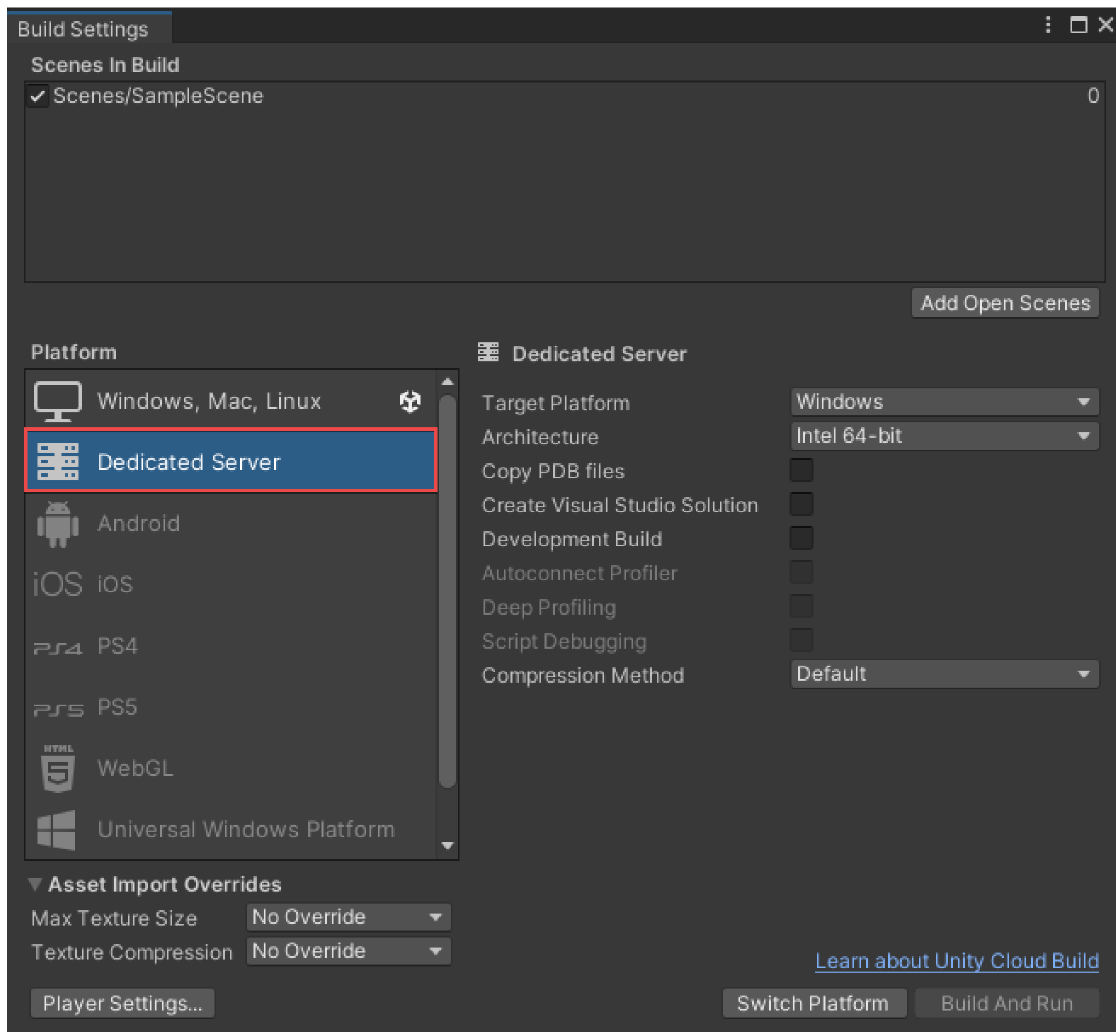


Fig. 25. Elección del modo de *build* en el editor de Unity [37]

El núcleo de la ejecución de la parte red, como ya se ha explicado, es la instancia de *NetworkRunner* presente en cada servidor y cliente. Este se crea en la clase del script *NetworkRunnerHandler.cs* y en el caso del servidor se encargará de crear una sesión de juego que se encuentre dentro de un *Lobby*. En Photon Fusion un *Lobby* es un contenedor de una lista de sesiones de juego entre las que un jugador puede filtrar para buscar por características específicas [21], en este caso, por una sesión con un mapa específico. En el caso de este prototipo existe un *Lobby* denominado como "EU", enfocado a contener sesiones de juego cuyo servidor dedicado se encuentre en Europa.

El prototipo está compuesto por dos escenas, *Menu* y *Game*. La escena *Menu* corresponde al menú del usuario, por lo que es una interfaz de usuario y no es relevante para el lado del servidor. Por tanto, en el *build* del servidor dedicado se seleccionará únicamente la escena *Game*, que contiene toda la lógica y la parte red del juego. El *build* para los clientes, en cambio, seleccionará ambas escenas al disponer de interfaz gráfica.

```
Task InitializeServerNetworkRunner(NetworkRunner runner, NetAddress address, Action<NetworkRunner> initialized, string[] sessionProperties)
{
    var sceneManager = runner.GetComponents(typeof(MonoBehaviour)).OfType<INetworkSceneManager>().FirstOrDefault();

    if (sceneManager == null)
    {
        //Handle networked objects that already exits in the scene
        sceneManager = runner.gameObject.AddComponent<NetworkSceneManagerDefault>();
    }

    runner.ProvideInput = true;

    var customProps = new Dictionary<string, SessionProperty>();

    customProps["map"] = (string)sessionProperties[1];

    return runner.StartGame(new StartGameArgs
    {
        GameMode = GameMode.Server,
        Address = address,
        SessionName = sessionProperties[0],
        SessionProperties = customProps,
        CustomLobbyName = "EU",
        PlayerCount = 5,
        Initialized = initialized,
        SceneManager = sceneManager
    });;
}
```

Fig. 26. Método para crear una sesión de juego en el prototipo de shooter multijugador en tercera persona

En la **Figura 26** se puede observar el método con el que un servidor dedicado crea una sesión de juego en el *Lobby* “EU”. El método se ejecuta en forma de tarea (*Task*) y lo primero que hace será inicializar el *NetworkSceneManager*, el cual sirve para el manejo de escenas y para sincronizar las escenas entre el servidor y los distintos clientes. Después se inicializa y se le da un valor a la variable *customProps*, la cual guarda las características personalizables de la sesión, en este caso el tipo de mapa que habrá. Por último, el *NetworkRunner* se encargará de crear e iniciar la sesión de juego en el lobby “EU” (argumento *CustomLobbyName*), con una capacidad para 5 jugadores (argumento *PlayerCount*) y con el nombre de sesión y mapa proporcionados por el array *sessionProperties* (argumentos *SessionName* y *SessionProperties*). El resto de los argumentos vienen dados por el *SceneManager* (inicializado al inicio del método), la dirección IP de la sesión (parámetro *address* recibido por la tarea), el modo en el que inicia la sesión (argumento *GameMode* inicializado en modo servidor) y el argumento para indicar si se inicializa la sesión de juego al crearla (*Initialized*).

Esta *Task* se ejecuta en el método *Start()* de la clase, tal como se observa en la **Figura 27**.

```
if (SystemInfo.graphicsDeviceType == GraphicsDeviceType.Null)
{
    string[] sessionProperties = Utility.readSessionProperties("Session properties.txt");
    mapGenerator.mapType = sessionProperties[1];
    mapGenerator.GenerateMap();

    var serverTask = InitializeServerNetworkRunner(networkRunner, NetAddress.Any(), null, sessionProperties);
    StartCoroutine(CheckServerConnection(serverTask));
    Debug.Log($"Server NetworkRunner iniciado.");
}
```

Fig. 27. Método *Start()* de la clase *NetworkRunnerHandler* del prototipo de shooter multijugador en tercera persona

En el método *Start()*, se llama al método *InitializeServerNetworkRunner* para crear e iniciar la sesión de juego y se realiza una comprobación de que se ha realizado la tarea correctamente mediante la corutina *CheckServerConnection* que pondrá un mensaje de error en el log y cerrará la aplicación del servidor si no se ha finalizado la tarea correctamente en 10 segundos. El valor del array *sessionProperties* se obtiene mediante el método *readSessionProperties* de la clase estática *Utility*, el cual se encarga de leer el fichero txt pasado como parámetro y retorna un array de *strings* con las dos primeras

líneas del fichero (cada línea es un elemento del array). Ese fichero deberá estar en la misma carpeta que el ejecutable del servidor dedicado y deberá contener el nombre de la sesión en la primera línea y el tipo de mapa que va a haber en la sesión en la segunda (Mapa 1, 2, 3, 4 y 5), tal como se puede ver en el ejemplo de la **Figura 28**. También se generará el tipo de mapa indicado en el fichero de forma local en el servidor, que será el mismo que se generará en los clientes (mediante el método *GenerateMap()* de la instancia de la clase *MapGenerator*). Las diferencias entre los 5 mapas están en las dimensiones del mapa, el número de obstáculos y el color de los mismos.

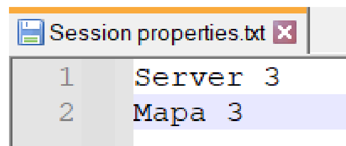


Fig. 28. Ejemplo de fichero txt para que el servidor dedicado cree la sesión de juego en el prototipo de shooter multijugador en tercera persona

Todo lo comentado en el método *Start()* se restringirá a su ejecución en los servidores mediante el condicional *SystemInfo.graphicsDeviceType == GraphicsDeviceType.Null*, con el cual se controla que únicamente ejecuten esa parte las ejecuciones que no tengan ningún tipo de gráficos, como es el caso del modo *Dedicated Server* de Unity.

La clase que implementa la interfaz *INetworkRunnerCallbacks*, cuyo funcionamiento ya se ha explicado en el apartado anterior, es la clase *Spawner*, que se encargará de gestionar las devoluciones de llamada del *NetworkRunner*. En esta clase, hay algunos métodos que se han orientado a que se ejecuten en el servidor, otros orientados a los clientes y otros a ambos. De momento, se van a explicar los que están orientados a que se ejecuten en el lado del servidor.

Los métodos de la clase *Spawner* que se ejecutan en el lado del servidor son *OnPlayerJoined* (se ejecuta cuando un jugador se une a la sesión de juego) y *OnPlayerLeft* (se ejecuta cuando un jugador abandona la sesión de juego).

```
public void OnPlayerJoined(NetworkRunner runner, PlayerRef player)
{
    if (runner.IsServer)
    {
        Transform spawnTransformPosition = map.GetRandomOpenTile();
        Vector3 spawnPosition = new Vector3(spawnTransformPosition.position.x, 1, spawnTransformPosition.position.z);
        NetworkObject networkPlayerObject = runner.Spawn(_playerPrefab, spawnPosition, Quaternion.identity, player);
        spawnedCharacters.Add(player, networkPlayerObject);
    }
}

public void OnPlayerLeft(NetworkRunner runner, PlayerRef player)
{
    if (spawnedCharacters.TryGetValue(player, out NetworkObject networkObject))
    {
        runner.Despawn(networkObject);
        spawnedCharacters.Remove(player);
    }
}
```

Fig. 29. Métodos de la interfaz *INetworkRunnerCallbacks* para ser ejecutados en el lado del servidor del prototipo shooter multijugador en tercera persona

En la **Figura 29**, se ven los dos métodos mencionados de la clase *Spawner*. En el *OnPlayerJoined* se restringe la ejecución a sólo el servidor mediante la llamada al método booleano *NetworkRunner.IsServer*, que retornará true si el *NetworkRunner* se está

ejecutando en modo servidor. Después, se obtendrá una posición aleatoria del mapa en la que no haya obstáculos mediante el método *GetRandomOpenTile* de la clase *MapGenerator* (que es la clase en la que se genera el mapa) y se generará un personaje en la posición aleatoria para el jugador que se ha unido (se asocia con su id de referencia y se le permite enviar inputs sobre el personaje, para moverle, por ejemplo) mediante el método *NetworkRunner.Spawn*, que es una adaptación de Photon Fusión del método *Instantiate* de Unity. La diferencia es que *Spawn* genera un *NetworkObject* visible y sincronizado para todos los jugadores y *Instantiate* sólo genera un objeto de forma local en el cliente o servidor donde se llama al método. Se utiliza el *prefab Player* como objeto a instanciar, que es un *NetworkObject* que representa el personaje de cada usuario y el cual se explicará más adelante en la parte del cliente. Por último, la id de referencia del usuario y el personaje creado se añadirán a un diccionario llamado *spawnedCharacters*.

Respecto al otro método (*OnPlayerLeft*), llamado cuando un usuario abandona la sesión, se buscará al usuario en el diccionario *spawnedCharacters* (el cual sólo tiene entradas en el lado del servidor) y se eliminará a su *NetworkObject* (su personaje) del mundo del juego de todos los usuarios y se borrará la entrada del diccionario. Por tanto, las acciones de crear y eliminar el personaje de un usuario sólo las puede llevar a cabo el servidor, evitando así posibles trampas.

Tras describir la parte del servidor, se va a explicar como se ha llevado a cabo la implementación de la parte del cliente. Lo primero que se encuentra al ejecutar el juego obtenido mediante el *build* para usuarios es un menú (escena *Menu* en Unity), en el que podrá elegir el mapa (desde el mapa 1 al mapa 5) en el que quiere jugar mediante un despegable o abrir el menú de opciones, como se puede ver en la **Figura 30.**

Fig. 30. Menú inicial del prototipo de shooter multijugador en tercera persona

En la **Figura 31** se puede ver el menú de opciones, el cual ofrece la posibilidad de ajustar el sonido y la resolución del juego, así como volver al menú de inicio. Todo ello ha sido implementado mediante barras de deslizamiento (*scrollbars*) y *toggles* (las casillas, de las

cuales solo puede haber una marcada), que son *GameObjects* correspondientes a la sección de *UI* (interfaz de usuario).

Fig. 31. Menú de opciones del prototipo de shooter multijugador en tercera persona

No se va a entrar en detalle en la implementación de esta parte, ya que corresponde a una interfaz de usuario que no es en lo que se centra este trabajo. La implementación se encuentra en el script *Menu.cs* (donde están programados los métodos que activan cada uno de los objetos de *UI* en la escena) y en el Canvas de la escena *Menu* del proyecto de Unity. Es una modificación de la escena *Menu* del proyecto que se ha utilizado como base del prototipo, lo cual se ha comentado al inicio de este apartado [17].

El prototipo está dotado tanto de música como de efectos de sonido (en ambas escenas), los cuales se introducen en la escena mediante un *GameObject* que contiene un *Audio Listener* (solo se puede tener un *Audio Listener* por escena) y los scripts *AudioManager.cs (*gestión del volumen y de los métodos para reproducir los sonidos), *MusicManager.cs (*gestión de la música al haber un tema para el menú y otro para la partida) y *SoundLibrary.cs* (guarda los efectos de sonido).

Una vez el jugador selecciona uno de los 5 mapas en el desplegable y pulsa sobre el botón "JUGAR" se cargará la nueva escena denominada como *Game*, la cual contiene todos los elementos para la partida. Una vez cargada la escena, se iniciará el objeto con el script *NetworkRunnerHandler.cs*, el cual se encargará de inicializar la instancia de *NetworkRunner* del cliente y de buscar una sesión de juego disponible con el mapa seleccionado en el desplegable. Todo ello se realizará mediante el procedimiento que establece Photon Fusion con su *Matchmaking API* [21] y mediante lo aprendido en el episodio *Online multiplayer FPS Unity & Photon Fusion EP7 (Lobby Session Browser)* [26] de la lista de reproducción sobre Photon Fusion. Además, también se generará el mapa seleccionado de forma local para el usuario, siendo el mismo mapa que se ha generado en el servidor y en todos los clientes de esa sesión. Todas estas acciones se realizarán entre el script *NetworkRunnerHandler.cs* y algunos métodos del script *Spawner.cs*, el cual se encuentra en el objeto *Spawner*, presente en la escena *Game* por

defecto. Dentro del *NetworkRunnerHandler.cs* se llama al método de la **Figura 32**, el cual, mediante la ejecución de una tarea, se une al *Lobby* “EU”.

```
Task JoinLobby(NetworkRunner runner)
{
    return runner.JoinSessionLobby(SessionLobby.Custom, "EU");
}
```

Fig. 32. Método de unión e un cliente a un *Lobby* en el prototipo de shooter multijugador en tercera persona

Al unirse el jugador al *Lobby* o al haber modificaciones en la lista de sesiones del *Lobby*, se ejecuta automáticamente el método *OnSessionListUpdated* (al que le llegan como parámetros la lista de sesiones y el *NetworkRunner* del jugador) de la interfaz *INetworkRunnerCallbacks* implementada en la clase *Spawner*. La implementación que se ha realizado de este método se puede ver en la **Figura 33**.

```
public void OnSessionListUpdated(NetworkRunner runner, List<SessionInfo> sessionList)
{
    if (SystemInfo.graphicsDeviceType != GraphicsDeviceType.Null && !runner.IsConnectedToServer)
    {
        if (sessionList.Count == 0)
        {runner.
            StartCoroutine(ConnectionError("Error de conexión: todos los servidores están caídos en este momento. Vuelva a
        }

        else
        {
            SessionInfo session = null;

            foreach (var sessionItem in sessionList)
            {
                // Check for a specific map
                if (sessionItem.Properties.TryGetValue("map", out var mapType) && mapType.IsString)
                {
                    var gameType = (string)mapType.PropertyValue;

                    // Check for the desired Game Map
                    if (mapType == PlayerPrefs.GetString("map"))
                    {
                        // Store the session info
                        session = sessionItem;
                        break;
                    }
                }
            }
            // Check if there is any valid session
            if (session != null)
            {
                Debug.Log($"Joining {session.Name}");
                var clientTask = InitializeClientNetworkRunner(runner, session);
                StartCoroutine(CheckConnection(clientTask));
            }

            else
            {
                StartCoroutine(ConnectionError("Error de conexión: ningún servidor con el mapa seleccionado disponible. Comienza
            }
        }
    }
}
```

Fig. 33. Implementación del método *OnSessionListUpdated* de la interfaz *INetworkRunnerCallbacks* en el prototipo de shooter multijugador en tercera persona

Debido al primer condicional if que aparece en el método *OnSessionListUpdated*, el código sólo podrá ser ejecutado por clientes que todavía no se han conectado a un servidor, es decir, clientes que están dentro del *Lobby,* pero fuera de una sesión de juego. La primera parte del código verificará que existan sesiones en el *Lobby* y en caso contrario

iniciará la ejecución de una corutina la cual muestra al usuario por pantalla el error pasado como parámetro y le envía al menú unos segundos después. En la **Figura 34** se puede ver un ejemplo de uno de los errores. En caso de haber sesiones, se iterará en la lista de sesiones hasta encontrar una sesión en cuyas propiedades se encuentre el mapa seleccionado por el usuario ene le menú. En caso de no encontrar ninguna sesión disponible con ese mapa, se iniciará una corutina para mostrar el error correspondiente al usuario y mandarle al menú. Si se encuentra una sesión, el usuario se unirá a esta mediante el método *InitializeClientNetworkRunner* (situado también dentro de *Spawner.cs*), cuya implementación se muestra en la **Figura 35**. Al igual que con el servidor, se ejecuta una corutina para comprobar que se une correctamente a la sesión de juego (corutina *CheckConnection*), mediante la cual, si el usuario no ha finalizado de unirse a la sesión en 5 segundos, se le mostrará al usuario el error de conexión correspondiente y se le enviará al menú.

Fig. 34. Ejemplo de error de conexión mostrado al usuario en el prototipo de shooter multijugador en tercera persona

En la implementación del método *InitializeClientNetworkRunner*, primero se carga el *NetworkSceneManager* del cliente y, a continuación, se ejecutará el proceso de unión cona la llamada del *NetworkRunner* del cliente a *StartGame*, pasándole como argumentos el modo cliente, el nombre de la sesión enviado por el método *OnSessionListUpdated* y el gestor de escenas cargado al principio del método. No se específica una dirección IP, por lo que no tendrá en cuenta la dirección IP del servidor para unirse.

```
Task InitializeClientNetworkRunner(NetworkRunner runner, SessionInfo session)
{
    var sceneManager = runner.GetComponents(typeof(MonoBehaviour)).OfType<INetworkSceneManager>().FirstOrDefault();

    if (sceneManager == null)
    {
        //Handle networked objects that already exits in the scene
        sceneManager = runner.gameObject.AddComponent<NetworkSceneManagerDefault>();
    }

    runner.ProvideInput = true;

    // Join
    return runner.StartGame(new StartGameArgs()
    {
        GameMode = GameMode.Client,
        SessionName = session.Name,
        Address = NetAddress.Any(),
        SceneManager = sceneManager,
    });
}
```

Fig. 35. Implementación del método para unir a un cliente a una sesión de juego en el prototipo de shooter multijugador en tercera persona

Al unirse a la sesión de juego, como se ha explicado antes, el servidor generará el personaje del usuario, sobre el que este únicamente tendrá autoridad para inputs, siendo el resto de las acciones sobre el *NetworkObject* del personaje exclusivas para el servidor al tener la autoridad de estado sobre este. El personaje se trata de un *prefab* cuya *skin* está formada por una cápsula y la cual contiene un arma formada por figuras geométricas (con *sprites* integrados para simular los efectos visuales de disparar), tal como se ve en la **Figura 36**. Dentro de los objetos contenidos en el *prefab Player*, los componentes más relevantes a parte del propio *NetworkObject* para que esté presente en la parte red con su propia ID sincronizado en el servidor y todos los clientes, están los scripts *Player.cs*, *HPHandler.cs* y *GunController.cs*.

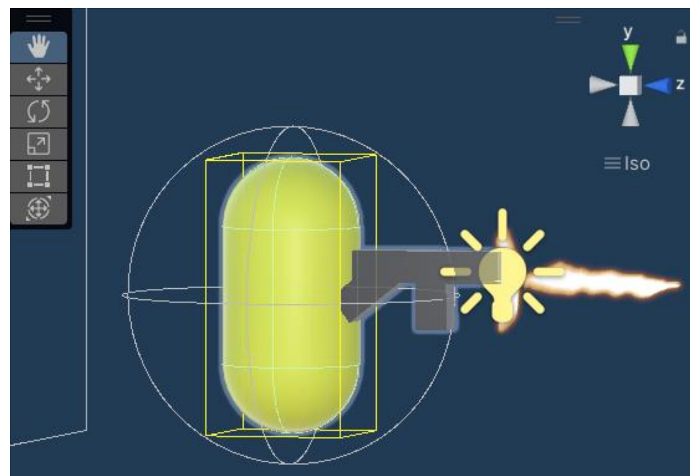


Fig. 36. Diseño del *prefab* del personaje de cada usuario en el prototipo de shooter multijugador en tercera persona

En la **Figura 37** se visualiza un estado de juego tras unirse a una sesión de juego de un servidor correctamente. En ese caso hay otros dos jugadores en la sesión con los que podrá jugar.

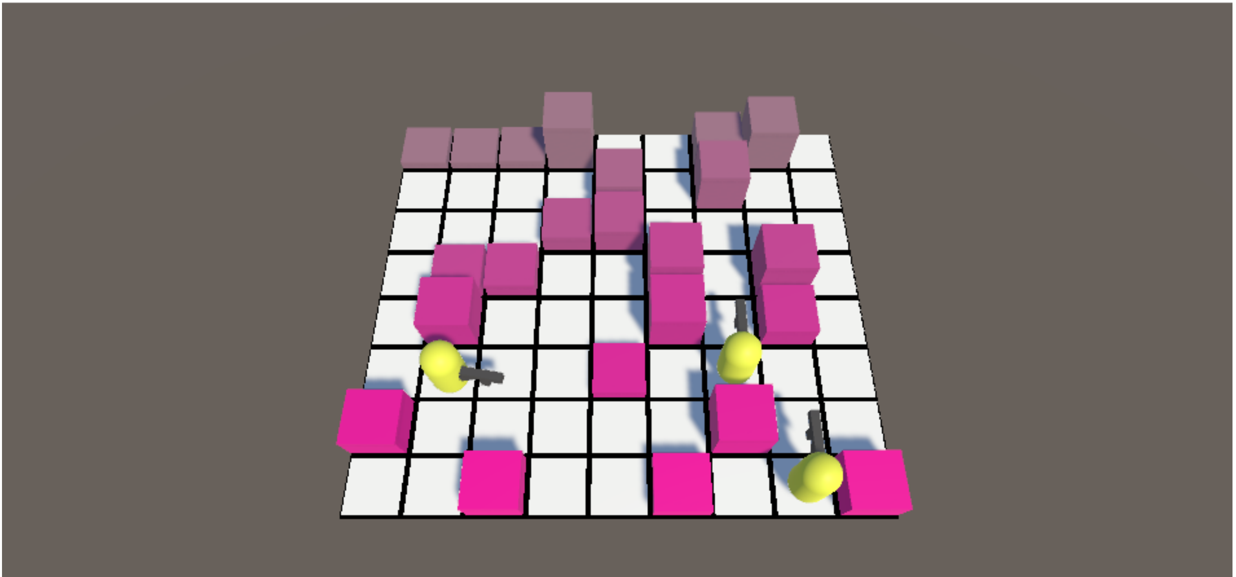


Fig. 37. Ejemplo de estado de juego tras unirse a una sesión de juego en el prototipo de shooter multijugador en tercera persona

La función principal de la clase *Player* será la de ejecutar las acciones correspondientes en base a los inputs que introduzca el jugador por teclado. El proceso de implementación del sistema de inputs para el movimiento y las acciones del jugador ha tomado como base la propia documentación de Photon Fusion [38] y lo aprendido en el *Multiplayer FPS Unity & Photon Fusion tutorial Project* [23]. Este sistema funciona mediante una serie de pasos, pudiendo ver primeros en la **Figura 38**:

- Primeramente, en la clase *Spawner* del cliente, se comprueba en cada fotograma si el usuario ha pulsado alguna tecla en el método *Update()*. En el caso de este prototipo, se comprueba si se han pulsado el botón izquierdo del ratón (utilizado para disparar), el botón derecho del ratón (utilizado para recargar el arma) o la barra espaciadora (utilizada para que el personaje salte). El resultado de la comprobación se guarda en una variable booleana para cada elemento.

- Lo siguiente será implementar el método *OnInput* de la interfaz *INetworkRunnerCallbacks* en la clase *Spawner*. Este método se encargará de captar el movimiento del personaje en el eje horizontal y vertical (ya sea mediante flechas o mediante las teclas ASDW) y lo guardará en un Vector3. Después, guardará su valor y el de las variables booleanas de comprobación del resto de inputs en una estructura, la cual se puede ver en la **Figura 39**. Esta estructura, la cual contiene los inputs del usuario, se enviará a Photon Fusion al darle el valor de la estructura creada a la estructura de Fusion *NetworkInput* (presente como parámetro en el método) mediante el método *Set*.

```
void Update()
{
    mouseButton0 = mouseButton0 | Input.GetMouseButton(0);
    mouseButton1 = mouseButton1 | Input.GetMouseButton(1);
    spaceButton = spaceButton | Input.GetKey(KeyCode.Space);
}
public void OnInput(NetworkRunner runner, NetworkInput input)
{
    var data = new NetworkInputData();

    Vector3 moveInput = new Vector3(Input.GetAxisRaw("Horizontal"), 0, Input.GetAxisRaw("Vertical"));
    data.direction += moveInput;

    if (mouseButton0)
    {
        data.buttons |= NetworkInputData.MOUSEBUTTON1;
    }
    if (mouseButton1)
    {
        data.buttons |= NetworkInputData.MOUSEBUTTON2;
    }
    if (spaceButton)
    {
        data.spaceButton = true;
    }

    mouseButton0 = false;
    mouseButton1 = false;
    spaceButton = false;

    input.Set(data);
}
```

Fig. 38. Sistema de inputs del prototipo de shooter multijugador en tercera persona

```
public struct NetworkInputData : INetworkInput
{
    public const byte MOUSEBUTTON1 = 0x01;
    public const byte MOUSEBUTTON2 = 0x02;
    public byte buttons;

    public bool spaceButton;
    public Vector3 direction;
}
```

Fig. 39. Estructura para enviar los inputs a Photon Fusion en el prototipo de shooter multijugador en tercera persona

Al tener las inputs guardados en Fusion, en el método *FixedUpdateNetwork()* de la clase *Player* (visible en la **Figura 40**), se obtendrá la estructura mediante el método de Fusion *GetInput* y en base a lo que haya guardado (que teclas o botones ha pulsado el usuario) se ejecutarán las acciones correspondientes (moverse, disparar, recargar o saltar). Las acciones de moverse y saltar vienen programadas por defecto por Photon Fusion en su clase *NetworkCharacterControllerPrototype* (variable _cc en el método), el cual es un componente del prefab del personaje. Mediante los métodos *Move* y *Jump* de esta clase, el personaje se moverá o saltará, apareciendo de forma sincronizada en todos los estados de juego del resto de clientes.

```
public override void FixedUpdateNetwork()
{
    if (GetInput(out NetworkInputData data))
    {
        data.direction.Normalize();
        _cc.Move(5 * Runner.DeltaTime * data.direction);

        if (transform.position.y < -10 && !respawnRequested)
        {
            GetComponent<HPHandler>().OnTakeDamage(GetComponent<HPHandler>().GetHP());
        }

        if (respawnRequested)
        {
            respawnRequested = false;
            FindObjectOfType<Spawner>().RespawnPlayer(this);
            hpHandler.OnRespawned();
        }

        if ((data.buttons & NetworkInputData.MOUSEBUTTON1) != 0 && !hpHandler.isDead)
        {
            gun.Shoot();
        }

        else if ((data.buttons & NetworkInputData.MOUSEBUTTON2) != 0 && !hpHandler.isDead)
        {
            gun.Reload();
        }

        if (data.spaceButton && !hpHandler.isDead)
        {
            _cc.Jump();
        }
    }
}
```

Fig. 40. Método para la ejecución de acciones del personaje en base a inputs en el prototipo de shooter multijugador en tercera persona

En el método de la **Figura 40** existe también una comprobación a una variable booleana, la cual indica si el jugador debe ser regenerado (es decir, si ha muerto). En caso afirmativo, ejecutará el método para regenerar el personaje, implementado en la clase *Spawner* y representado en la **Figura 41**. En este método se obtiene una posición aleatoria del mapa libre de obstáculos y se coloca al personaje en ella mediante el método *TeleportToPosition* que se encuentra en la clase *NetworkTransform* (la alternativa de *Transform* para la red de Photon Fusion) [24]. En el *FixedUpdateNetwork()* de la clase *Player* se comprueba también si el personaje se ha salido de los límites del mapa, en cuyo caso se le reducirá la vida a 0 llamando al método *OnTakeDamage* de la clase *HPHandler* (la cual solicitará la regeneración del personaje una vez muere dándole el valor True a la variable *respawnRequested*), cuyo funcionamiento se explicará más adelante.

```
public void RespawnPlayer(Player player)
{
    Transform spawnTransformPosition = map.GetRandomOpenTile();
    Vector3 spawnPosition = new Vector3(spawnTransformPosition.position.x, 1, spawnTransformPosition.position.z);
    player._cc.TeleportToPosition(spawnPosition);
}
```

Fig. 41. Método para regenerar a un personaje de un usuario en el prototipo de shooter multijugador en tercera persona

Respecto a los métodos para disparar (*Shoot()*) y recargar (*Reload()*), no están incluidos en el controlador de movimiento integrado por Photon Fusion, por lo que se han

programado en la clase *GunController*, la cual contiene la implementación de todas las acciones del arma del personaje.

```
public void Shoot()
{
    if (!isReloading && Time.time > nextShotTime && projectilesRemainingInMag > 0)
    {
        for (int i = 0; i < projectileSpawn.Length; i++)
        {
            if (projectilesRemainingInMag == 0)
            {
                break;
            }
            projectilesRemainingInMag--;
            nextShotTime = Time.time + msBetweenShots / 1000;
            StartCoroutine(ShootEffect());
            Projectile newProjectile = Runner.Spawn(_prefabProjectile,
                projectileSpawn[i].position, projectileSpawn[i].rotation,
                Object.InputAuthority, (runner, o) => {
                    o.GetComponent<Projectile>().Init(muzzleVelocity);
                });
        }
    }
}
```

Fig. 42. Método con la funcionalidad de disparo del arma en el prototipo de shooter multijugador en tercera persona

En la **Figura 42**, se puede ver la implementación de la acción de disparar con el arma. El primer condicional *if* del método permite que sólo se ejecute la acción en caso de que no se esté recargando el arma, haya pasado el tiempo suficiente desde el último disparo y queden balas en el cargador del arma. En caso de que se cumplan todas las condiciones, se generarán los proyectiles que se hayan estipulado en la variable *projectileSpawn* (en este caso 2) mediante el método *Spawn* de Photon Fusion. Cada proyectil está basado en el prefab *Bullet*, cuyos complementos son *NetworkObject* (para convertirlo en un objeto en red visible para el resto de los usuarios), *NetworkTransform* (para que se sincronice su movimiento) y el script *Projectile.cs*, que va a ser explicado a continuación. Además, está dotado de un objeto 3D en forma de ortoedro para simular la forma de una bala y de un *Trail Renderer* para representar la estela que deja el trayecto un disparo. En el método de disparo se inicia también una corutina que se encarga de generar una serie de efectos visuales y de sonido para dotar de más detalle a esta acción, la cual se puede ver reflejada en la **Figura 43**.

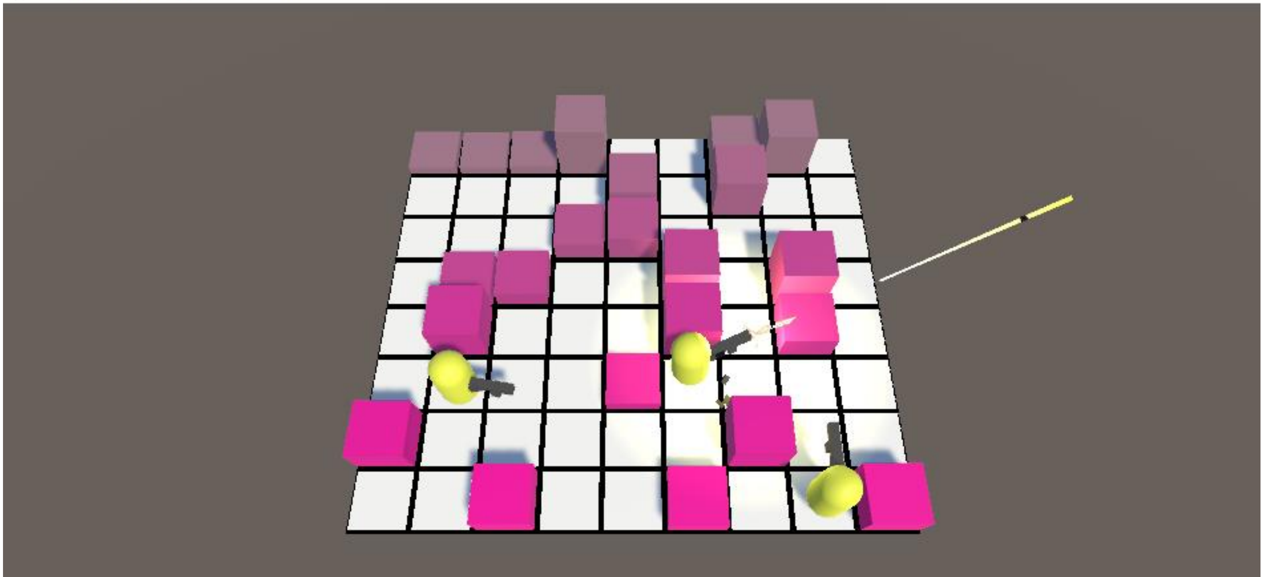


Fig. 43. Acción de disparo en el prototipo de shooter multijugador en tercera persona

Respecto a la acción de recargar, simplemente activará una animación de recarga y asignará a la variable con el número de balas del cargador el número inicial (la capacidad máxima del cargador del arma). En caso de que el arma del personaje se quede sin balas, se ejecutará el método de recarga automáticamente (además de poder recargar en cualquier momento mediante el botón derecho del ratón).

Una vez creado cada proyectil, su la lógica de su comportamiento estará definida por la clase *Projectile*.

```
public override void FixedUpdateNetwork()
{
    transform.position += speed * transform.forward * Runner.DeltaTime;

    if (Object.HasStateAuthority)
    {
        if (life.Expired(Runner))
        {
            Runner.Despawn(Object);
        }
        else
        {
            float moveDistance = speed * Runner.DeltaTime;
            CheckCollisions(moveDistance);
        }
    }
}

1 referencia
void CheckCollisions(float moveDistance)
{
    if (Runner.LagCompensation.Raycast(transform.position, transform.forward, moveDistance, Object.InputAuthority, out var hitinfo, collisionMask))
    {
        hitinfo.Hitbox.transform.root.GetComponent<HPHandler>().OnTakeDamage(1);
    }
}
```

Fig. 44. Implementación del comportamiento de un proyectil en el prototipo de shooter multijugador en tercera persona

En la **Figura 44** se puede ver la implementación del comportamiento de cada proyectil. Tras crearse, el proyectil se moverá en la dirección en la que ha sido generado a una velocidad determinada (determinada en la variable *Speed*). Tras 5 segundos, el proyectil se destruirá, lo cual está determinado por la variable *life*, que es una variable *TickTimer* (estructura integrada por Fusion que sirve para implementar una cuenta atrás) y con la

propiedad *Networked* aplicada para que el contador de los 5 segundos tenga el mismo valor en todos los estados de jugo de los clientes y el servidor.

Durante los 5 segundos de vida, el proyectil comprobará en cada *tick* si ha impactado contra otro personaje con el método *CheckCollisions*. Para implementar esto, se ha hecho uso de la característica de compensación de latencia (*LagCompensation*) integrada por Photon Fusion, la cual implementa la técnica de compensación de latencia de asistencia de control. Para ello, se hace uso del componente *Hitbox* (aplicado al objeto "Hitbox", presente en el prefab del personaje) y el *Hibox Root* (aplicado en el nodo padre del prefab del personaje). Al incluir estos componentes en el personaje de cada usuario, el método *LagCompensation.RayCast* permitirá al servidor revisar el historial de posiciones del *Hitbox* de cada cliente y decidir si se produce el impacto mediante la realización de *raycasts* (comprobación de colisiones mediante rayos) en la visión del mundo del cliente en el momento en el que ha realizado el input en vez de en la del servidor, ya que el estado del mundo del cliente será anterior al del servidor debido a la latencia. De esta forma la comprobación será mucho más precisa, ya que será específica para la situación de cada cliente, evitando así inconsistencias en el juego [39].

Por ejemplo, el cliente podría disparar y según su estado del juego alcanza con una bala al personaje de otro usuario, mientras que en el estado de juego del servidor (que va más avanzado), ese personaje ya se ha movido y no se produce el impacto del disparo. Al comprobar el impacto desde el punto de vista del usuario, el impacto se produce y se evita que la latencia afecte a la acción del disparo.

En caso de impactar con un personaje, se le restará un punto de vida mediante la llamada al método *OnTakeDamage* de la clase *HPHandler* del usuario impactado por el proyectil. Además, el color del personaje se volverá blanco durante un breve periodo de tiempo, tal como se muestra en la **Figura 45**.

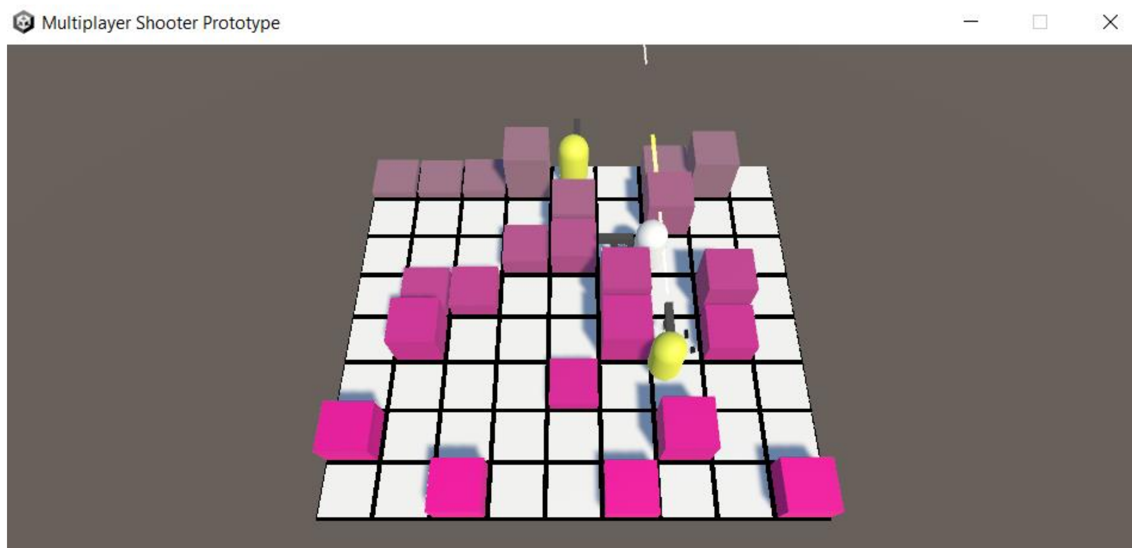


Fig. 45. Situación de impacto de un disparo en el prototipo de shooter multijugador en tercera persona

El script *HPHandler.cs* tiene implementado la gestión de la salud del personaje y del estado de este, manejando factores como la solicitud de *respawn* al morir. En esta clase se hace uso de la propiedad *Networked* de Photon Fusion y de su funcionalidad para vincular métodos con el evento de cambio de valor de la variable, ejecutándose así en

cada cambio de valor. El uso de esta propiedad se implementa para los atributos *HP* (guarda la salud del usuario, la cual comienza en 5) y *isDead* (variable booleana que indica si el personaje está muerto) para que así su valor sea el mismo para la instancia del personaje en el estado de juego de todos los clientes.

```
[Networked(OnChanged = nameof(OnHPChanged))]
9 referencias
byte HP { get; set; }

[Networked(OnChanged = nameof(OnStateChanged))]
11 referencias
public bool isDead { get; set; }

IEnumerator ServerReviveCO()
{
    yield return new WaitForSeconds(2.0f);
    GetComponent<Player>().RequestRespawn();
}


//Function only called on the server
2 referencias
public void OnTakeDamage(byte damage)
{
    //Only take damage while alive
    if (isDead)
        return;

    HP -= damage;

    Debug.Log($"{Time.time} {transform.name} took damage got {HP} left ");

    //Player died
    if (HP <= 0)
    {
        Debug.Log($"{Time.time} {transform.name} died");

        StartCoroutine(ServerReviveCO());

        isDead = true;
    }
}
```

Fig. 46. Método para gestionar los impactos de disparos en el prototipo de shooter multijugador en tercera persona

En la **Figura 46** se encuentra la declaración de los atributos *HP* y *isDead* mencionados y la vinculación de su cambio de valor a los métodos *OnHPChanged* y *OnStateChanged,* además de la implementación del método *OnTakeDamage* al que se llama cada vez que el objeto del personaje de un usuario (*Player*) recibe el impacto de un proyectil. Este método resta el valor de daño recibido por el proyectil (establecido en 1 en el prototipo) al valor de la salud actual del personaje y comprueba si después de eso la salud es 0 o menor, en cuyo caso inicia la ejecución de una corutina en la que cambia el valor de la variable *respawnRequested* de la clase *Player* a True para que se pueda ejecutar el método *RespawnPlayer* de la clase *Spawner* que transportará al usuario a un punto aleatorio disponible del mapa (se ha explicado y visto ya este método). Cada vez que la salud del

personaje varía, se ejecuta el método *OnHPChanged*, el cual cambiará momentáneamente el color del material del personaje a blanco (como se ha visto antes) y también pondrá en rojo la pantalla del usuario durante un breve tiempo.

La última acción del método *OnTakeDamage* será cambiar el valor de la variable *isDead* a True, lo cual activará la ejecución del método *OnStateChanged*, visible en la **Figura 47**.

```
static void OnStateChanged(Changed<HPHandler> changed)
{
    Debug.Log($"{Time.time} OnStateChanged isDead {changed.Behaviour.isDead}");

    bool isDeadCurrent = changed.Behaviour.isDead;

    //Load the old value
    changed.LoadOld();

    bool isDeadOld = changed.Behaviour.isDead;

    //Handle on death for the player. Also check if the player was dead but is now alive in that case revive the player.
    if (isDeadCurrent)
        changed.Behaviour.OnDeath();
    else if (!isDeadCurrent && isDeadOld)
        changed.Behaviour.OnRevive();
}
```

Fig. 47. Método para gestionar la muerte de un personaje en el prototipo de shooter multijugador en tercera persona

Tal como se observa en la **Figura 47**, la implementación de los métodos vinculados a un cambio en el valor de una variable *Networked* se lleva a cabo mediante el uso de un parámetro del tipo *Changed* que se introducido por Fusión. Este tipo guarda el valor actual y el valor anterior de la variable (accesible mediante el método *LoadOld*), por lo que se puede llevar a cabo una comparación entre los distintos valores para llevar a cabo una acción u otra. El valor de la variable (en este caso *isDead*) se obtiene mediante el uso de *Behaviour*."nombreVariable". En el caso del método *OnStateChanged*, cada vez que se ejecuta se comprueba si el personaje está muerto actualmente, en cuyo caso se ejecutará el método *OnDeath*, que generará un efecto de sonido y otro visual de partículas (visible en la **Figura 48**) y desactivará el objeto del personaje, haciendo que no sea visible. Una vez el método *RespawnPlayer* de la clase *Spawner* haya finalizado su ejecución, cambiará de nuevo el estado de *isDead* a False. En este caso el método *OnStateChanged*, entrará en el condicional *else if* que comprueba si el jugador está vivo actualmente, pero estaba muerto en el estado anterior. El código dentro de este condicional ejecutará el método *OnRevive*, el cual volverá a activar el objeto *Player*, volviendo a ser visible para todos los usuarios desde la nueva posición aleatoria establecida.

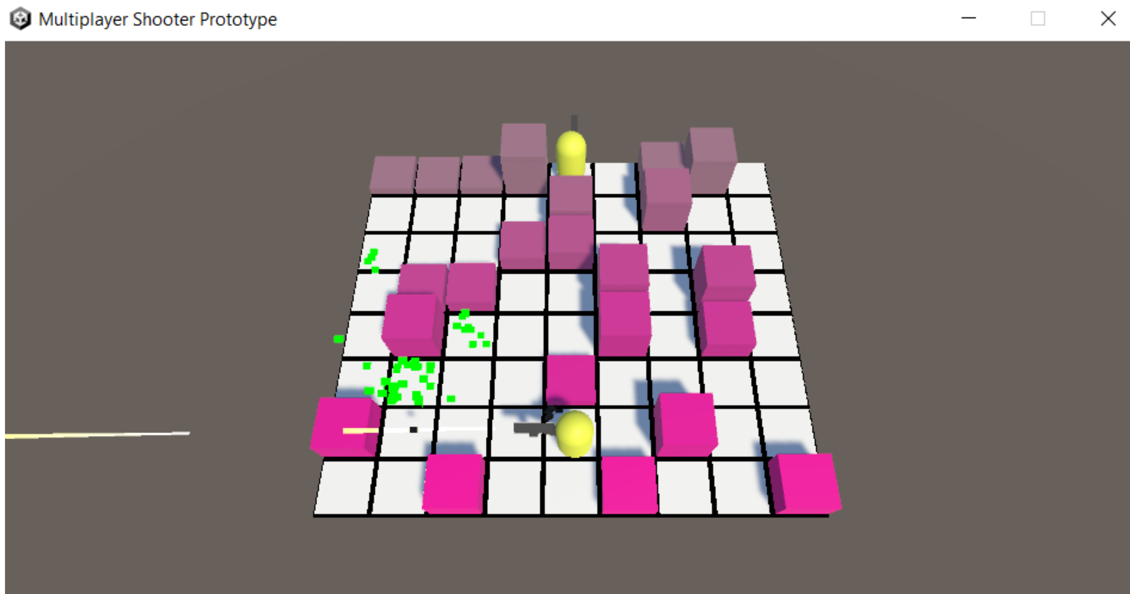


Fig. 48. Muerte de un personaje en el prototipo de shooter multijugador en tercera persona

Tanto la parte de implementación de los proyectiles como la de la gestión de la salud del personaje, se han basado en parte en lo aprendido en el vídeo *Tutorial: Online multiplayer FPS Unity & Photon Fusion EP3 (shooting + health)* [25].

El último script que resta por explicar es el *GameUI.cs*, el cual implementa la interfaz de usuario dentro de la partida, mostrando al principio de la partida el nombre del mapa al que se ha unido el usuario y el número de jugadores máximo, tal como se observa en la **Figura 49**.

Fig. 49. Interfaz de usuario al inicio de la partida en el prototipo de shooter multijugador en tercera persona

Para terminar de explicar la fase de desarrollo de este prototipo, se va a describir como se ha implementado la segunda técnica de compensación de latencia utilizada: la interpolación. Esta técnica reduce la inestabilidad visual que puede producir la latencia mediante la predicción de estados pasados perdidos, basándose en el estado de juego actual y el anterior, recuperando, por tanto, estados intermedios entre esos dos para que el juego se vea mucho más fluido para el usuario, evitando cambios visuales bruscos que puedan empeorar la experiencia de juego para el usuario debido a la latencia.

La implementación de la interpolación en Photon Fusion viene integrada por defecto, haciendo que sea un *framework* muy completo como ya se ha comentado. La forma de activar esta técnica para el personaje del usuario (objeto *Player*) se muestra en la **Figura 50**. En este caso se configura en el componente del controlador de movimiento, para el cual se introduce el modelo creado para el personaje (la cápsula amarilla con el arma) en el campo *Interpolation Target*, que es el campo donde se introduce a lo que se desea aplicar la interpolación. El resto de los campos sirven para configurar los rangos de error de conexión, posición y rotación que se quieren aplicar.

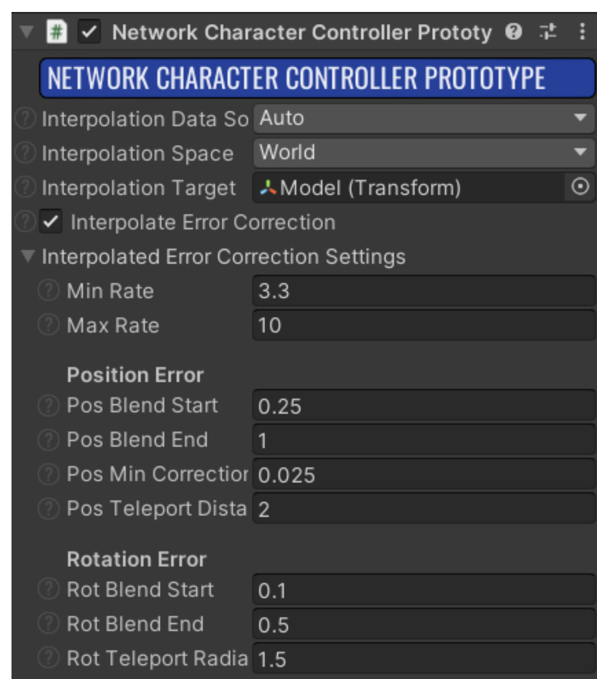


Fig. 50. Configuración de la interpolación en el controlador de movimiento del personaje en el prototipo de shooter multijugador en tercera persona

La aplicación de la interpolación en los proyectiles (prefab *Bullet*) se implementa de la misma forma, solo que el modelo de la bala se introducirá en el campo *Interpolation Target* del componente *NetworkTransform* al no existir un controlador de movimiento.

5.2. Implementación del prototipo de videojuego de cartas multijugador basado en turnos

El segundo prototipo de videojuego en red que se ha implementado es un videojuego de cartas multijugador (cuyo funcionamiento se ha explicado en la fase de diseño), el cual se basará en turnos y dispondrá de partidas para dos usuarios. Aunque ya dispongo de una

base de conocimiento en Unity aprendida mediante la realización del primer prototipo, se ha utilizado una herramienta de red distinta, por lo que hay que realizar una formación previa en la herramienta Netcode for GameObjects, creada por Unity.

5.2.1. Fase de formación en las herramientas a utilizar

La formación previa al desarrollo del prototipo se basa principalmente en el aprendizaje de las características que ofrece la herramienta Netcode for GameObjects para implementar la parte red de un videojuego y en como implementar esas características.

Debido a que este trabajo de fin de grado se centra en la parte red de los videojuegos, se ha utilizado el arte de las cartas, el diseño del tablero y las funcionalidades de robar (*Draw*) y barajar cartas (*Shuffle*) de un tutorial de YouTube que lo proporcionaba [27].

La formación en el uso de la herramienta de Unity Netcode for GameObjects se ha llevado a cabo principalmente del vídeo *COMPLETE Unity Multiplayer Tutorial (Netcode for Game* Objects*)* [28]. Es una herramienta que tiene muchas similitudes con Photon Fusion, siendo las siguientes las utilizadas en este prototipo:

- Existe también el componente *Network Object* teniendo la misma funcionalidad que en Fusion, es decir, hacer el objeto visible en la red para el resto de los usuarios y para el servidor.
- Se utiliza la propiedad de variable *NetworkVariable* en sustitución de la propiedad *Networked* utilizada en Photon Fusión, teniendo ambas la misma funcionalidad (valor único para la instancia del objeto en red para todos los estados de juego del resto de clientes y el servidor).
- Se utiliza el *NetworkManager* en sustitución del *NetworkRunner* como clase para gestionar la ejecución de la parte red del cliente o servidor.
- Parte de la formación se ha enfocado en la implementación de las RPC (llanadas remotas procedurales) mediante NetCode for Gameobjects, ya que se utilizan en este prototipo.
- La implementación de los métodos para la devolución de llamadas de la interfaz *INetworkRinnrrCallbacks* de Photon Fusion se hace mediante métodos ya integrados en la clase *NetworkManager* de Netcode for GameObjects.
- Gestión de trampas: el servidor dedicado de la sesión tendrá la autoridad exclusiva sobre todos los objetos de red. Los clientes sólo podrán realizar modificaciones en las secciones clave de la parte red mediante el uso de las RPC, por lo que será el servidor el que lleve a cabo esas modificaciones, haciendo uso así de la técnica antitrampas de no confiar en el jugador.
- Técnicas de compensación de latencia: en este prototipo no es relevante la latencia al no afectar en la experiencia de juego, por lo que la formación no se ha centrado en ello.

Para el desarrollo de la parte de matchmaking en este prototipo, se necesita hacer uso de los denominados Unity Gaming Services, ya que Netcode for GameObjects no tiene integrada esta funcionalidad. Esta plataforma ofrece a los desarrolladores de videojuegos funcionalidades alojadas en la nube con las que implementar o perfeccionar aspectos de los videojuegos en red. Para este prototipo ha sido necesaria la formación en el servicio de Unity Relay y el de Unity Lobby, con los cuales se ha implementado la parte de

matchmaking del prototipo de videojuego multijugador de cartas. También se ha hecho uso del servicio de Unity Authentication, ya que es necesario que cada servidor o cliente esté autenticado para hacer uso de estos servicios. Se ha implementado la autenticación anónima para que no se tenga que realizar un inicio de sesión.

Unity Relay es un servicio que, entre otras opciones, permite implementar un sistema seguro de conectividad de confianza entre los jugadores y el servidor, utilizando un servidor de *Relay* universal que actúa como proxy entre el servidor y los clientes [31].

Respecto al servicio en la nube Unity Lobby, se utiliza para la gestión de sesiones de juego. Este servicio permite que los clientes pueden buscar sesiones mediante un código, realizar búsquedas filtradas (por mapa, modo de juego…), crear y buscar sesiones privadas, realizar una búsqueda rápida de sesión para unirse a cualquiera que tenga disponibilidad (está funcionalidad es la que se ha implementado en este prototipo), etc. [32]. A diferencia de Photon Fusion, en Netcode for GameObjects un lobby no representa una lista de sesiones de juego si no que representa una sesión de juego.

Para aprender como funcionan estos servicios, se ha hecho uso del tutorial *How to setup Global Matchmaking for Unity*, el cual muestra un ejemplo de implementación del matchmaking en un videojuego en red desarrollado con Netcode y los servicios de Unity Gaming Services de Relay y Lobby complementos entre sí [29].

5.2.2. Fase de desarrollo del prototipo de videojuego en red

La explicación de la fase de desarrollo del prototipo de videojuego multijugador de cartas va a comenzar por la parte del servidor dedicado. Al igual que en el primer prototipo, se ha utilizado el módulo de *Dedicated Game Server* de Unity, el cual permite tener un *build* distinto al del cliente, generando un ejecutable sin parte gráfica (que no es necesaria en el servidor) para reducir el consumo de recursos y optimizar la ejecución del servidor.

La única escena del prototipo, llamada *Game*, está compuesta por diversos elementos gráficos referentes a la interfaz de usuario (que sólo se utilizarán en el ejecutable de los clientes) y al tablero, las 10 cartas distintas disponibles representadas por *GameObjects* con un *sprite* con el arte de la carta y el script *Card.cs* con la lógica del funcionamiento de la carta, el objeto *GameManager* con el script *GameManager.cs* (que se explicará en la parte cliente) y el objeto *Network Manager* compuesto por el script *NetworkManager cs* (al que se le añade como atributo el prefab *NetworkEntity*, que será el objeto que se generará al iniciar la ejecución del *NetworkManager*) y el componente Unity Transport.

El script *NetworkManager.cs,* es el encargado de gestionar las acciones del *Network Manager* (explicado en la fase de formación del prototipo) tanto del servidor como de los clientes, es decir gestiona la ejecución de la parte red. En la parte del servidor de esta clase (la del cliente se explicará posteriormente), se crea la sesión de juego inicializando las instancias de *Relay* y *Lobby* a las que se conectarán los clientes. Para implementar esta parte referente al matchmaking se ha hecho uso en parte del script extraído del tutorial mencionado en la fase de formación [30]. También se gestionará la desconexión de clientes de la sesión de juego.

```
private async Task<Lobby> CreateLobby()
{
    try
    {
        const int maxPlayers = 3;

        // Create a relay allocation and generate a join code to share with the lobby
        var a = await RelayService.Instance.CreateAllocationAsync(maxPlayers);
        var joinCode = await RelayService.Instance.GetJoinCodeAsync(a.AllocationId);

        // Create a lobby, adding the relay join code to the lobby data
        var options = new CreateLobbyOptions
        {
            Data = new Dictionary<string, DataObject> { { JoinCodeKey, new DataObject(DataObject.VisibilityOptions.Public, joinCode) } }
        };

        StreamReader txtFile = new StreamReader("Session name.txt");
        string sessionName = txtFile.ReadLine();
        txtFile.Close();
        var lobby = await Lobbies.Instance.CreateLobbyAsync(sessionName, maxPlayers, options);

        // Send a heartbeat every 15 seconds to keep the room alive
        StartCoroutine(HeartbeatLobbyCoroutine(lobby.Id, 15));

        // Set the game room to use the relay allocation
        transport.SetHostRelayData(a.RelayServer.IpV4, (ushort)a.RelayServer.Port, a.AllocationIdBytes, a.Key, a.ConnectionData);

        // Start the room
        NetworkManager.Singleton.StartServer();
        NetworkManager.Singleton.OnClientDisconnectCallback += RestartGame;
        return lobby;
    }
    catch (Exception e)
    {
        Debug.Log($"Error crando el lobby: {e}");
        return null;
    }
}
```

Fig. 51. Método que implementa la creación de una sesión de juego en el prototipo de videojuego de cartas multijugador basado en turnos

En la **Figura 51** se puede observar el método *CreateLobby*, mediante el cual el servidor creará la sesión de juego. Para ello, primeramente, se crea la instancia del servicio de Relay (método *CreateAllocationAsync*) y se genera el código (método *GetJoinCodeAsync*) mediante el cual los clientes se conectarán al Relay del servidor (este código no se introducirá manualmente, sino que lo obtendrán los clientes directamente mediante el servicio de Lobby). Después, se creará la sesión de juego (mediante el método *CreateLobbyAsync* del servicio de Lobby) a la que se unirán los clientes, estableciendo el número máximo de jugadores del *Lobby* en 3 (se cuenta también al servidor) y el nombre de la sesión escrito en un fichero de texto que debe estar en la carpeta del ejecutable. A continuación, se iniciará la ejecución de la corutina *HeartBeatLobbyCoroutine*, que servirá para enviar un pulso cada 15 segundos al *Lobby* para que no se cierre.

Tras esto se determinarán las características del objeto de Unity Transport (objeto para definir características de la capa de transporte tales como la dirección IP, el puerto, características del *Relay* definido…) del servidor, que serán las de la sala de juego. Por último, una vez está ya todo configurado, se inicia la ejecución del *NetworkManger* del servidor mediante el método *StartServer* y se vincula el método que gestiona las desconexiones de clientes al evento *ClientDisconnectCallback*, lo cual hará que se ejecute el método en el servidor cada vez que haya una desconexión de un cliente.

Antes de llamar al método *CreateLobby* el servidor habrá ejecutado el método de autenticación visible en la **Figura 52**. En este método se inicializan los servicios de Unity Gaming Services para el dispositivo que lo ejecuta mediante el uso de *UnityServices.InitializeAsync* y después se auténtica de forma anónima para evitar tener que iniciar sesión mediante la ejecución del método *SignInAnonymouslyAsync* del

servicio Authentication. Este método de Netcode, crea una autenticación única para el dispositivo, por lo que no se pueden llevar a cabo dos ejecuciones simultáneas en la misma máquina (saltaría un error). Para poder realizar pruebas, se ha utilizado la herramienta de Unity ParrelSync, la cual permite crear clones del proyecto en otra ventana del editor. Para evitar que autentique a ambos clones como si estuviesen en el mismo dispositivo, se utiliza el método *options.SetProfile* cuando la ejecución se lleva a cabo en un editor de Unity en vez de mediante un archivo exe para que la id de autenticación (guardada en la variable *playerId*) sea distinta en cada clon.

```
    private async Task Authenticate()
    {
        var options = new InitializationOptions();

#if UNITY_EDITOR
        // It's used to differentiate the clients in the different Unity editor windows (using ParrelSync), otherwise lobby will count them as the same
        options.SetProfile(ClonesManager.IsClone() ? ClonesManager.GetArgument() : "Primary");
#endif

        await UnityServices.InitializeAsync(options);

        await AuthenticationService.Instance.SignInAnonymouslyAsync();
        playerId = AuthenticationService.Instance.PlayerId;
    }
```

Fig. 52. Método para implementar la autenticación el prototipo de videojuego de cartas multijugador basado en turnos

El método *RestartGame* presente en la **Figura 53** es el que se ejecutará en el servidor cada vez que exista una desconexión de un cliente. Este método utilizará los métodos del *NetworkManager* del servidor para comprobar si hay algún jugador más en la partida a parte del que se ha desconectado, en cuyo caso llamará al método *RestartClientGame* de la instancia presente en el servidor del objeto del personaje (*NetworkEntity*) del jugador restante. Ese método (que se verá más adelante) restablecerá la salud y el tablero del jugador a la situación al inicio de la partida, ya que este se quedará esperando a la unión de otro jugador para iniciar una nueva partida en esa sesión.

```
void RestartGame(ulong disconnectClientId)
{
    if (NetworkManager.Singleton.ConnectedClientsIds.Count > 0)
    {
        ulong opponentPlayerID;
        foreach (ulong clientID in NetworkManager.Singleton.ConnectedClientsIds)
        {
            if (clientID != disconnectClientId)
            {
                opponentPlayerID = clientID;
                NetworkEntity opponentPlayer = NetworkManager.Singleton.ConnectedClients[opponentPlayerID].PlayerObject.GetComponent<NetworkEntity>();
                opponentPlayer.RestartClientGame();
                break;
            }
        }
    }
}
```

Fig. 53. Método para gestionar la desconexión de los clientes en el prototipo de videojuego de cartas multijugador basado en turnos

La siguiente parte de la fase de desarrollo del prototipo que se va a tratar es la parte cliente de la clase *NetworkHandler*. En esta parte se implementa la búsqueda y unión a una sesión de juego y la gestión del abandono de partida de los jugadores (ya sea por voluntad propia o por fallos de conexión). Antes de unirse a una sesión de juego, el cliente realiza una llamada al método *Authenticate* explicado en la parte del servidor, mediante el cual se autenticará anónimamente y obtendrá su id único de usuario.

```
private async Task<Lobby> QuickJoinLobby()
{
    try
    {
        var lobby = await Lobbies.Instance.QuickJoinLobbyAsync();

        // If we found one, grab the relay allocation details
        var a = await RelayService.Instance.JoinAllocationAsync(lobby.Data[JoinCodeKey].Value);

        // Set the details to the transform
        SetTransformAsClient(a);

        // Join the game room as a client
        NetworkManager.Singleton.StartClient();
        NetworkManager.Singleton.OnClientDisconnectCallback += ServerDisconnection;
        return lobby;
    }
    catch (Exception e)
    {
        Debug.Log($"No hay lobbys disponibles: {e}");
        return null;
    }
}
```

Fig. 54. Método que implementa la unión a una sesión de juego para los jugadores en el prototipo de videojuego de cartas multijugador basado en turnos

En la **Figura 54** se puede observar la implementación del método *QuickJoinLobby*, mediante el cual los usuarios se unen a una partida en el juego. Para ello, se llamará al método *QuickJoinLobbyAsync()* de la clase *Lobbies* (del servicio Lobby), el cual retornará una sesión disponible (un objeto de tipo *Lobby*), que se guardará en la variable "lobby". Después, se recuperarán los datos del *Relay* del servidor que gestiona esa sesión mediante el método *JoinAllocationAsync*, al que se le pasará el código de unión obtenido mediante el lobby guardado previamente. Tras esto, se configurarán los parámetros del elemento Uniy Transport del cliente en el método *SetTransformAsClient*, en el cual mediante el método *SetClientRelayData* se añadirán las características de red obtenidas del *Relay* creado por el servidor previamente. Mediante esto se determina, entre otras cosas, el puerto y la dirección IP a las que el cliente se conectará para acceder a la sesión de juego. Es decir, el servicio de Relay contiene todas las características de red de la sesión (puerto, dirección IP, código de unión…) y el servicio de Lobby permite que los clientes obtengan ese *Relay* de forma cómoda sin necesidad de tener que introducir ningún código y siendo por tanto un proceso de conexión automático al iniciar el juego. Por último, se iniciará la ejecución el *NetworkManager* en modo cliente, el cual se conectará a la sesión mediante los datos introducidos en el elemento Unity Transport. También se vincula el método que gestiona la desconexión del servidor al evento *ClientDisconnectCallback* del *NetworkManager*, lo cual hará que se ejecute el método en el cliente cada vez que haya una desconexión del propio usuario por fallos de conexión (ya sea el servidor o el cliente el que haya perdido la conexión).

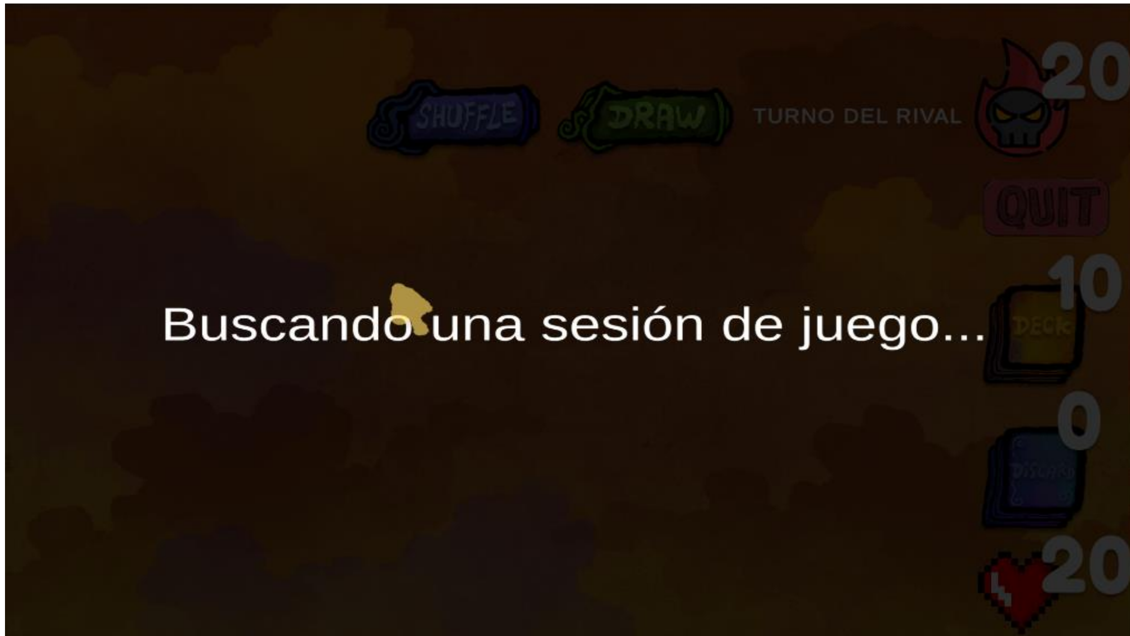


Fig. 55. Mensaje informativo al usuario mientras busca sesión en el prototipo de videojuego de cartas multijugador basado en turnos

Por tanto, el método *ServerDisconnection* se ejecutará en el cliente en caso de que el propio cliente se desconecte del servidor. Este método mostrará por pantalla un mensaje de error de conexión (como en la **Figura 55** pero con otro contenido de texto distinto) durante 5 segundos y tras esto, llamará al método *LeaveSession* mostrado en la **Figura 56**, el cual llevará a cabo varias acciones antes de proceder a cerrar el juego. Este método también es llamado cuando el usuario decide abandonar la partida voluntariamente cerrando la aplicación mediante la propia X de la ventana o pulsando el botón "QUIT" de la interfaz de usuario. También lo ejecutará el servidor en caso de cerrarse voluntariamente o por fallo de conexión.

Cuando un usuario o servidor sale de la sesión por el motivo que sea, es necesario sacarles de esta, ya que el servicio Lobby no lo hace automáticamente. Para ello, primeramente, se realiza una búsqueda del lobby al que corresponde la sesión para comprobar que siga existiendo (ya que el abandono de la sesión puede ser causa de la caída del servidor, en cuyo caso el lobby ya no existiría) mediante el uso del método *QueryLobbiesAsync*, al que se le pasa como parámetro los filtros que se quieren aplicar a la búsqueda (en este caso el nombre del lobby al que se había conectado el cliente en el método *QuickJoinLobby*). En caso de que la búsqueda retorne un resultado, significará que la sesión sigue activa, por lo que se procederá a realizar la acción correspondiente al abandono de la sesión:

- En caso de que la entidad que está realizando el abandono sea el host del lobby (el servidor dedicado en el caso de este prototipo), se eliminará la sesión de juego mediante la llamada al método *DeleteLobbyAsync* integrado por el servicio Lobby de los Unity Gaming Services.

- Si el que está abandonando la sesión es un cliente, se utilizará el método *RemovePlayerAsync* del servicio Lobby para eliminar al usuario, al que se le pasarán como parámetros el id de autenticación del usuario y el nombre del lobby del que se quiere eliminar al usuario.

Por último, para cerrar la ejecución del juego, se marca la variable booleana *UnityEditor.EditorApplication.isPlaying* como falsa para los casos d ejecución del juego en el editor de Unity o se llama al método *Application.Quit()* para los casos en los que se ejecuta el juego haciendo uso del archivo exe. El método *TryQuit* del que se desvincula al evento *Application.wantsToQuit* (evento que se ejecuta cuando se intenta cerrar el juego mediante la X de la ventana o mediante la llamada directa a *Application.Quit*), permite que antes de que se cierre la aplicación, se ejecute el método *LeaveSession* para poder salir de la sesión correctamente.

```
    private async void LeaveSession()
    {
        try
        {
            StopAllCoroutines();

            if (connectedLobby != null)
            {
                QueryLobbiesOptions options = new QueryLobbiesOptions();
                options.Count = 25;

                // Search if the lobby still exists
                options.Filters = new List<QueryFilter>()
                {
                    new QueryFilter(
                    field: QueryFilter.FieldOptions.Name,
                    op: QueryFilter.OpOptions.EQ,
                    value: connectedLobby.Name)
                };
                QueryResponse lobbies = await Lobbies.Instance.QueryLobbiesAsync(options);

                // If the lobby exists, remove the player from it or delete it in case of the server
                if (lobbies.Results.Count > 0)
                {
                    if (connectedLobby.HostId == playerId)
                    {
                        await Lobbies.Instance.DeleteLobbyAsync(connectedLobby.Id);
                    }
                    else
                    {
                        await Lobbies.Instance.RemovePlayerAsync(connectedLobby.Id, playerId);
                    }
                }
            }
        }
        catch (Exception e)
        {
            Debug.Log($"Error en la desconexión con el lobby: {e}");
        }

#if UNITY_EDITOR
        if (EditorApplication.isPlaying)
        {
            UnityEditor.EditorApplication.isPlaying = false;
        }
#else
        Application.wantsToQuit -= TryQuit;
        Application.Quit();
#endif
    }
```

Fig. 56. Método para gestionar los fallos de conexión en el prototipo de videojuego de cartas multijugador basado en turnos

Tanto para la ejecución de la *Task CreateLobby* en el servidor como para la del *QuickJoinLobby* en el cliente, se ejecuta en paralelo una corutina que comprueba si tras unos segundos se ha ejecutado la tarea correctamente y si no mostrará un error de conexión por pantalla (en el caso del cliente, informando de que no hay sesiones de juego disponibles) y cerrará la ejecución.

Este prototipo dispone de 10 cartas distintas que tendrá cada uno de los jugadores. Cada objeto de carta dispone de un valor de ataque, un valor de defensa (guardados junto a varios métodos en la clase *Card*) y un arte gráfico específico. El valor de ataque y defensa de una carta se mostrará al usuario sólo cuando este tenga el cursor sobre la carta (implementado esto en el método *OnMouseOver* de la clase *Card*), tal como se muestra en la **Figura 57**.

Fig. 57. Ejemplo de tablero del usuario con información sobre una carta en el prototipo de videojuego de cartas multijugador basado en turnos

El tablero inicial de la partida para cada jugador es igual que el de la **Figura 57** pero sin ninguna carta sobre el tablero, es decir, con 10 cartas en el "deck". Como se puede observar la interfaz muestra el número de vidas del jugador en la esquina inferior derecha, el número de vidas del oponente en la esquina superior derecha, el número de cartas en el "deck" para poder robar (hasta tener un máximo de 5 en el tablero) y el número de cartas en el montón de descartadas (ambos montones representados como listas de objetos tipo *Card* en el código del script *GameManager.cs*), que son las que se han jugado para atacar o defender. Además, se indica si es el turno del jugador o es el del oponente. También existen tres botones: botón "SHUFFLE", para poner las cartas descartadas en el "deck", botón "DRAW" para coger una carta del "deck" y ponerla sobre el tablero y botón "QUIT" para cerrar la ejecución del juego.

Si un usuario pulsa sobre una carta con el clic izquierdo del ratón, se ejecutará el método *OnMouseDown*, implementado en el script *Card.cs* y mostrado en la **Figura 58**. Este método, primeramente, comprueba si la carta no se ha jugado y si el usuario tiene permiso para un movimiento en ese momento (es decir, si le toca atacar o defender en ese momento). En caso de que se cumplan las dos condiciones, se generarán efectos visuales y de animación de la carta y se moverá hacia delante. Por último, se comprueba si es el

turno del usuario, en cuyo caso se ejecutará el proceso de ataque y en caso contrario el de defensa, presentes ambos en la clase *NetworkEntiy,* la cual se explicará más adelante.

```
private void OnMouseDown()
{
    if (!hasBeenPlayed && gm.movementTime)
    {
        hollowCircleInst = Instantiate(hollowCircle, transform.position, Quaternion.identity);

        camAnim.SetTrigger("shake");
        anim.SetTrigger("move");

        transform.position += Vector3.up * 3f;
        hasBeenPlayed = true;
        gm.movementTime = false;

        if (gm.player.IsTurn())
        {
            gm.player.Attack(this);
        }

        else
        {
            gm.player.Defend(this);
        }
    }
}
```

Fig. 58. Implementación del funcionamiento de una carta la pulsar sobre ella en el prototipo de videojuego de cartas multijugador basado en turnos

Al finalizar un turno, tanto la carta atacante como la defensora se moverán a su correspondiente montón de descartadas mediante la llamada al método *MoveToDiscardPile* de la clase *Card*, el cual se encargará de añadirlo a la lista de descartadas de la clase *GameManager*, de dejar un hueco libre en el tablero para poder robar nuevas cartas del “deck” y de desactivar el objeto correspondiente a esa carta para que no sea visible. Además, llamará al método *UITextUpdate* (utilizado para actualizar datos de la interfaz de usuario) de la clase *GameManager* para actualizar el número de cartas en el montón de descartadas que se muestra al usuario.

La siguiente parte que se va a explicar es la clase *GameManager*, la cual está orientada a gestionar las funciones de la partida que se ejecutan a nivel local en el cliente y no en la parte red, tales como las funcionalidades de los botones de la interfaz de usuario, el método para actualizar los datos mostrados en la interfaz y la gestión del final de la partida. También guarda las dos listas de los dos montones de cartas y gestiona los lugares disponibles en el tablero para las cartas, así como los lugares donde deben aparecer visibles.

La funcionalidad del botón “DRAW” está implementada en el método *DrawCard*, visible en la **Figura 59**. En caso de que haya como mínimo una carta en el “deck”, se generará un número aleatorio entre 0 y el tamaño de la lista *deck* - 1 mediante el uso de la clase de Unity *RandomNumberGenerator*. Después, se comprobará si hay algún espacio libre en el tablero, en cuyo caso se activará el objeto de la carta en ese espacio, haciéndose visible. También se eliminará la carta de la lista *deck* y se actualizará el dato de número de cartas en el “deck” presente en la interfaz mediante el método *UITextUpdate* (se restará uno).

```
public void DrawCard()
{
    if (deck.Count >= 1)
    {
        camAnim.SetTrigger("shake");

        RandomNumberGenerator.Create();

        Card randomCard = deck[RandomNumberGenerator.GetInt32(0, deck.Count)];
        for (int i = 0; i < availableCardSlots.Length; i++)
        {
            if (availableCardSlots[i] == true)
            {
                randomCard.gameObject.SetActive(true);
                randomCard.handIndex = i;
                randomCard.transform.position = cardSlots[i].position;
                randomCard.hasBeenPlayed = false;
                deck.Remove(randomCard);
                availableCardSlots[i] = false;
                UITextUpdate("deck", deck.Count);
                return;
            }
        }
    }
}
```

Fig. 59. Implementación de la funcionalidad de coger carta del "deck" en el prototipo de videojuego de cartas multijugador basado en turnos

Para la funcionalidad del botón "SHUFFLE" se ha implementado el método *Shuffle*, el cual añadirá las cartas presentes en la lista de descartadas (*discardPile*) a la lista del "deck", dejando la lista de descartadas vacía. Por último, se actualizan los datos del número de cartas en cada montón mediante el uso del método *UITextUpdate*.

Al pulsar sobre el botón "QUIT", se ejecutará el método *LeaveSession* (explicado previamente) para abandonar la sesión de forma correcta antes de cerrar el juego.

Como última funcionalidad a explicar de la clase *GameManager*, comentar el método *FinishGame*, al cual se llama desde la clase *NetworkEntity* cuando la vida de uno de los dos jugadores llega a 0 o menos. Este método recibe como parámetro una variable booleana que indicará si el jugador que está ejecutando el método ha ganado, en cuyo caso se mostrará un mensaje de victoria en la pantalla (en caso contrario se mostrará uno de derrota). También se ejecutará una corutina que intentará cerrar el juego tras 5 segundos, lo cual implica que se ejecute el método de *LeaveSession* justo antes de cerrar el juego debido a la vinculación del evento *Application.wantsToQuit* al método *TryQuit* en la clase *NetworkHandler*, como ya se ha explicado antes. De esta forma, los dos jugadores abandonarán la sesión y dejarán dos huecos libres para que otros jugadores inicien una nueva partida en esa misma sesión con el mismo servidor dedicado.

La explicación de la fase de desarrollo del prototipo, continua por los métodos implementados en la clase *NetworkEntity* (objeto que se crea con el único de la ejecución del *NetworkManager*). Este objeto representa la entidad de cada jugador y contiene el componente *Network Object*, por lo que habrá una instancia por jugador en cada mundo del juego (es decir, 2 instancias presentes en el servidor y en cada cliente). El script

NetworkEntity.cs forma parte del prefab *Network Entity* y ese prefab se ha añadido a la propiedad "PlayerPrefab" del objeto *NetworkManager*, lo cual implica que cada vez que un cliente inicie la ejecución de su *NetworkManager*, se genere su instancia de *NetworkEntity*, la cual representará al jugador y se iniciará una vez el cliente se haya conectado a una sesión de juego.

Esta clase está enfocada a gestionar la salud y el turno de los jugadores y a administrar las defensas y los ataques mediante la comunicación entre el servidor y los clientes mediante "Remote Procedure Calls" (RPC). La salud y el turno de cada jugador están representadas mediante las variables "health" (de tipo int y con valor inicia 20) y "turn" (de tipo bool e indica si es el turno del jugador), las cuales tienen aplicada la propiedad *NetworkVariable* para que su valor se sincronice para ser el mismo para esa instancia en los dos clientes y el servidor de la sesión. El valor de las variables con esta propiedad sólo puede ser modificado por el servidor, aplicando de esta forma la técnica antitrampas de no confiar en el jugador al no permitir que el código de los clientes modifique estas propiedades, que son de importancia clave para cada jugador.

La implementación de las RPC mediante Netcode for GameObjects debe seguir dos normas básicas de nomenclatura:

- Justo encima del método se debe indicar si se desea que se ejecute en el servidor (poniendo "[ServerRpc]") o si está enfocado a su ejecución en los clientes (poniendo "[ClientRpc]").
- El nombre del método debe terminar en "ServerRpc" si está enfocado a ejecutarlo en el servidor o en "ClientRpc" para ejecutarlo en los clientes.

Además, se puede hacer uso de los parámetros "ServerRpcParams" en los métodos para el servidor (utilizados, por ejemplo, para obtener el id del cliente que ha solicitado la ejecución del método) o "ClientRpcParams" (utilizados, por ejemplo, para indicar la id del cliente en el que se quiere ejecutar el método si no se desea ejecutarlo en todos).

Cada entidad (servidor o cliente) dispondrá en su mundo dos instancias de *NetworkEntity*, representando cada una a uno de los jugadores, aunque el cliente será propietario únicamente de su propia instancia. El cliente podrá saber cuál es su instancia mediante el método integrado *IsOwner* de la clase de Netcode *Network Ob*ject, el cual retornará "true" si el cliente es propietario del *Network Object*. Para diferenciar la ejecución del servidor dedicado de la de un cliente, se puede utilizar el método integrado *IsServer* de la clase *NetworkManager,* el cual retornará "true" si la entidad es un servidor.

Tras crear una instancia de *NetworkEntity*, lo primero que se lleva a cabo son los métodos *Start* y *OnNetworkSpawn*. El método *Start* realizará una RPC al servidor desde la instancia de la que el cliente sea propietario (la que representa a ese jugador) mediante el método *GetTurnServerRpc*. En el método *OnNetworkSpawn* se vinculan los métodos *HealthUpdate* y *TurnUpdate* a los eventos de cambio de valor (*OnValueChanged*) de las *NetworkVariables* "health" y "turn" mencionadas anteriormente. Esto último es una funcionalidad integrada por Netcode para las variables con la propiedad *NetworkVariable*. El método *OnNetworkSpawn* se ejecuta en todas las instancias de *NetworkEntity* de los dos clientes y del servidor. La corutina *InitConnectionStatus* ejecutada sólo en los clientes, muestra un mensaje en la pantalla del usuario sobre el estado en el que se encuentra el inicio de la partida (si se está buscando a un oponente que se una a la sesión, si se ha encontrado un oponente, si la partida va a comenzar…).

La llamada RPC al método del servidor *GetTurnServerRpc* (visible en la **Figura 60**), permitirá al servidor decidir el jugador que tendrá el turno inicial. Para ello, primero comprueba si el número de jugadores conectados a la sesión es 2 y, en ese caso, otorga el turno a ese jugador dándole el valor "true" a su variable "turn" y el valor "false" a la del otro jugador. Como ya se ha comentado, las variables con la propiedad *NetworkVariable* solo pueden ser modificadas por el servidor, por lo que se hace uso de las RPC para que los clientes soliciten las modificaciones y el servidor pueda comprobar que no hay uso de trampas antes de llevar a cabo las modificaciones solicitadas por el cliente. La id del cliente que llama al método se obtiene mediante los "ServerRpcParams", mientras que la del oponente se busca en la lista de clientes conectados en la sesión *(ConnectedClientIds*) presente sólo en el *NetworkManager* del servidor.

```
[ServerRpc]
1 referencia
private void GetTurnServerRpc(ServerRpcParams serverRpcParams)
{
    if (NetworkManager.Singleton.ConnectedClientsIds.Count == 2)
    {
        NetworkEntity player = NetworkManager.Singleton.ConnectedClients[serverRpcParams.Receive.SenderClientId].PlayerObject.GetComponent<NetworkEntity>();

        ulong opponentPlayerID = 0;
        foreach (ulong clientID in NetworkManager.Singleton.ConnectedClientsIds)
        {
            if (clientID != player.OwnerClientId)
            {
                opponentPlayerID = clientID;
                break;
            }
        }
        NetworkEntity opponentPlayer = NetworkManager.Singleton.ConnectedClients[opponentPlayerID].PlayerObject.GetComponent<NetworkEntity>();

        player.turn.Value = true;
        opponentPlayer.turn.Value = false;
    }
}
```

Fig. 60. Implementación del método para obtener el turno inicial en el prototipo de videojuego de crtas multijugador basado en turnos

El método *TurnUpdate* será ejecutado cada vez que la variable "turn" cambie de valor, incluido el turno inicial. Este tipo de métodos vinculados al evento *OnValueChanged*, deben tener dos parámetros del tipo de la variable a la que están vinculados: el primer parámetro guarda el valor previo de la variable y el segundo el valor nuevo asignado. En la **Figura 61** se puede ver la implementación de este método, el cual se ejecuta antes del turno inicial y al final de cada turno, ya que esas son las ocasiones en las que se cambia el valor de la variable.

Lo primero que se realiza en el método es la destrucción de la carta del oponente ("opponentCard"). Esa carta corresponde a la carta que ha utilizado el jugador oponente en el turno (ya sea para defender o atacar), la cual se genera de forma local cuando el oponente la elige (se instancia una copia de esa carta) y se destruye al final del turno. Después, se ejecutará el segundo bloque de código del método, el cual sólo se ejecuta en la instancia de *NetworkEntity* que representa al propio jugador (de la que es propietario). En este bloque se hace uso del parámetro *newTurn*, correspondiente al nuevo valor de la variable "turn". En caso de ser true, significa que el jugador ha sido defensor en el último turno, por lo que mueve al montón de descartadas la carta guardada en la variable "defenderCard" (la variable "attackerCard" tendrá el valor null). En caso contrario se hará lo mismo, pero con la carta guardada en la variable "attackerCard". Se comprueba si estas variables son null para que en el turno inicial no se ejecute esta parte al no haber habido ningún movimiento todavía. Por último, se llamará al método *UITextUpdate* de la clase *GameManager* para actualizar el texto de la situación del turno en la interfaz del usuario.

Para el jugador que tiene el turno para atacar en el siguiente turno, se dará el valor “true” a su variable que indica que tiene permiso de movimiento (*movementTime* de la clase *GameManager*) para que el jugador pueda pulsar sobre alguna de sus cartas del tablero.

```
void TurnUpdate(bool previousTurn, bool newTurn)
{
    if (opponentCard != null)
    {
        Destroy(opponentCard.gameObject);
    }

    if (IsOwner)
    {
        if (newTurn)
        {
            if (defenderCard != null)
            {
                defenderCard.MoveToDiscardPile();
                defenderCard = null;
            }
            gm.UITextUpdate("turn", 1);
            gm.movementTime = true;
        }

        else
        {
            if (attackerCard != null)
            {
                attackerCard.MoveToDiscardPile();
                attackerCard = null;
            }
            gm.UITextUpdate("turn", 0);
        }
    }
}
```

Fig. 61. Implementación del método que se ejecuta en los cambios de turno de los jugadores en el prototipo de videojuego de cartas multijugador basado en turnos

El método *HealthUpdate* se ejecuta con las modificaciones de la salud de los jugadores, es decir, tras la resolución de cada ataque con la correspondiente defensa. Esté método se encargar de actualizar los datos de las vidas de la interfaz de usuario, tanto para el oponente como para el propio jugador mediante la llamada al método *UITextUpdate* de *GameManager*. Para ello, en la instancia de la que es propietario el jugador modificará el dato de su propia salud, mientras que a otra instancia se encargará de modificar los datos de salud del oponente en la interfaz. Para ello, se hace uso del nuevo valor de la variable “health”, ya que su valor está sincronizado entre todos los clientes. También, para ambos casos, se llamará al método *FinishGame* de la clase *GameManager* en caso de que la nueva salud tenga valor de 0 o menor. Debido a que este método se ejecuta dos veces en cada entidad (una por cada instancia), se podrá diferenciar si es la instancia del oponente la que se ha quedado sin vidas (en cuyo caso se cargará la pantalla informativa de victoria) o la instancia de la que es propietario del jugador (se cargaría la pantalla informativa de derrota).

En la clase *NetworkEntity* está implementada la forma de restablecer el tablero y las propiedades de un jugador para los casos en los que el oponente se desconecta de la partida. Cuando esto sucede, como ya se ha explicado, el servidor desde la clase *NetworkHandler* llama al método *RestartClientGame* de la clase *NetworkEntity*. Este método se encarga de restablecer el valor de la salud del usuario a 20 (ya que sólo puedo

hacer modificaciones el servidor) y ejecutará el método *RestartBoardClientRpc* de forma remota en el cliente. Este último método, presente en la **Figura 62**, se ejecuta mediante RPC en la parte del cliente debido a que se encarga de hacer modificaciones a nivel local del cliente (el servidor restablece la parte red y el cliente la parte local). Se encargará de restablecer a null las variables con las cartas defensora y atacante, de mover todas las cartas del tablero al montón de descartas y eliminar la copia de la carta defensora o atacante del rival (en caso de que se haya desconectado a mitad de un turno). Por último, ejecutará el método *Shuffle* de la clase *GameManager* para que las 10 cartas estén en el "deck" y restablecerá el dato de la interfaz del número de vidas del oponente a 20 (el dato de su propio número de vidas se habrá modificado en la ejecución del método *HealthUpdate* al haberse restablecido previamente el número de vidas desde el servidor). También se mostrará un mensaje por pantalla al usuario indicando la desconexión del oponente y mostrándole que se está esperando a otro jugador. Cuando un jugador nuevo se una a la sesión, se iniciará una nueva partida desde cero.

```
[ClientRpc]
1 referencia
private void RestartBoardClientRpc()
{
    if (IsOwner)
    {
        attackerCard = null;
        defenderCard = null;

        Card[] cardsInHand = FindObjectsOfType<Card>(false);
        foreach (Card card in cardsInHand)
        {
            if (card.transform.parent != null)
            {
                card.MoveToDiscardPile();
            }
            else
            {
                Destroy(card.gameObject);
            }
        }

        gm.Shuffle();
        gm.UITextUpdate("opponent health", 20);
        gm.connectionInfo.SetActive(true);
        gm.connectionResult.text = "El jugador rival se ha desconectado. Esperando a un jugador para una nueva partida...";
    }
}
```

Fig. 62. Implementación del método para restablecer la parte local del cliente en el prototipo de videojuego de cartas multijugador basado en turnos

La última parte por explicar de la fase de desarrollo del prototipo de videojuego de cartas multijugador basado en turnos es la forma en la que se ha implementado el proceso de acciones en cada turno, es decir, la fase de ataque y defensa y la resolución de estos dos movimientos. Ambos jugadores tendrán habilitados los botones durante cualquier momento de la partida. Al principio del turno, el jugador del que es el turno tendrá la variable "movementTime" con valor verdadero, por lo que podrá pulsar sobre cualquiera de sus cartas para atacar. Al pulsar sobre una carta, se ejecutará el método *Attack*, el cual guardará la carta en la variable "attackerCard" y realizará una RPC al servidor, el cual ejecutará el método *SendAttackServerRpc* visible en la **Figura 63**.

En este método el servidor recibe como parámetros desde el cliente el valor de ataque de la carta, el nombre del *GameObject* correspondiente a esa carta y los "ServerRpcParams" para que el servidor sepa que cliente ha realizado la llamada. Con esto, el servidor buscará en su propio estado del juego la carta con ese mismo nombre (el servidor tiene en su

mundo las mismas 10 cartas que ambos jugadores) y comprobará que tiene el mismo valor de ataque que el que ha enviado el cliente, aplicado así la técnica antitrampas de no confiar en el jugador, ya que se impide que el cliente pueda mandar un valor de ataque falso. En caso de que el valor sea el mismo, el servidor procede a realizar una RPC al otro cliente (el defensor) mediante el método *GetOpponentCardClientRpc*. La llamada sólo se realiza a ese cliente al hacer uso de los "ClientRpcParams", ya que si no la llamada se hace a todos los clientes por defecto.

```
[ServerRpc]
1 referencia
private void SendAttackServerRpc (int damage, string card, ServerRpcParams serverRpcParams)
{
    Card serverCard = GameObject.Find(card).GetComponent<Card>();

    if (serverCard.attack == damage)
    {
        gm.attackerDamage = damage;
        ulong attackerPlayerID = serverRpcParams.Receive.SenderClientId;
        ulong defenderPlayerID = 0;
        foreach (ulong clientID in NetworkManager.Singleton.ConnectedClientsIds)
        {
            if (clientID != attackerPlayerID)
            {
                defenderPlayerID = clientID;
                break;
            }
        }

        GetOpponentCardClientRpc(card, true, new ClientRpcParams { Send = new ClientRpcSendParams { TargetClientIds = new List<ulong> { defenderPlayerID } } });
    }
}
```

Fig. 63. Implementación de la gestión del ataque por el servidor en el prototipo de videojuego de cartas multijugador basado en turnos

El jugador defensor en ese turno recibirá como parámetros el nombre de la carta atacante y un booleano con valor "true", el cual indica que el jugador está recibiendo un ataque. El método *GetOpponentCardClientRpc* (mostrado en la **Figura** 64) permite al jugador instanciar a carta utilizada por el oponente en ese turno para visualizarla. Para ello, busca la carta con ese mismo nombre en su estado de juego y la instancia creando una copia en la posición del "opponentCardSlot", establecida en la esquina superior izquierda, tal como se ve en la **Figura 65**. La última acción implementada en este método es la asignación del valor verdadero a la variable "movementTime" del objeto *GameManager* para que el jugador pueda realizar el movimiento de defensa. Tal como se ha visto en el método *OnMouseDown* de la clase *Card*, cuando un jugador pulsa una carta, el valor de la variable "movementTime" de ese jugador cambiará a "false" para que ese jugador no pueda activar el movimiento de más cartas hasta el siguiente turno. Esta última acción solo se realizará en caso de que el cliente que esté ejecutando el método sea el defensor en ese turno.

```
[ClientRpc]
2 referencias
private void GetOpponentCardClientRpc(string card, bool receiveAttack, ClientRpcParams clientRpcParams)
{
    Card[] allCards = FindObjectsOfType<Card>(true);
    Card opCard = null;

    foreach (Card searchCard in allCards)
    {
        if (searchCard.gameObject.name == card)
        {
            opCard = searchCard;
            break;
        }
    }

    opponentCard = Instantiate(opCard);
    opponentCard.gameObject.SetActive(true);
    opponentCard.transform.position = opponentCardSlot.position;
    opponentCard.hasBeenPlayed = true;

    if (receiveAttack)
    {
        gm.movementTime = true;
    }
}
```

Fig. 64. Implementación del método para generar la carta utilizada por el oponente en el prototipo de videojuego de cartas multijugador basado en turnos

Fig. 65. Situación de tablero en la que un jugador recibe un ataque en el prototipo de videojuego de cartas multijugador basado en turnos

Por tanto, tras esto, el jugador defensor podrá pulsar sobre cualquiera de sus cartas del tablero para defender, llamando al método *Defend* al no ser su turno. Este método guardará la carta pulsada en la variable "defenderCard" y realizará una RPC al servidor, el cual ejecutará el método *SendDefenseServerRpc* visible en la **Figura 67**.

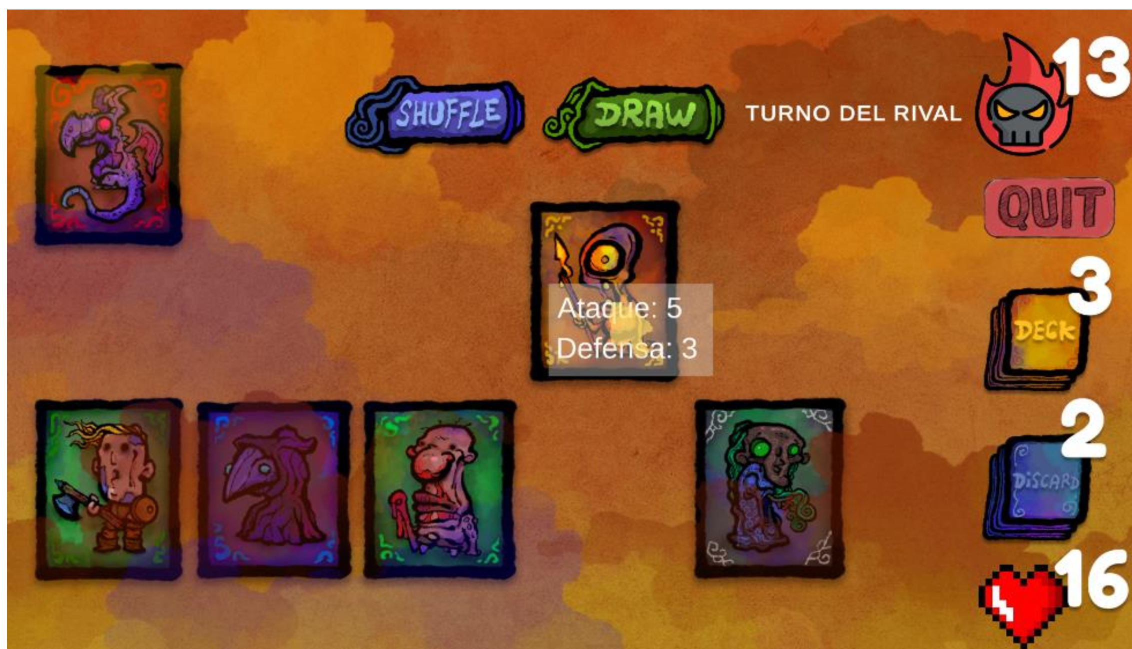


Fig. 66. Situación de tablero en la que un jugador realiza una defensa en el prototipo de videojuego de cartas multijugador basado en turnos

En este método el servidor recibe como parámetros desde el cliente el valor de defensa de la carta, el nombre del *GameObject* correspondiente a esa carta y los "ServerRpcParams" para que el servidor sepa que cliente ha realizado la llamada. De la misma manera que en el método del servidor para gestionar el movimiento de ataque, el servidor buscará la carta con ese mismo nombre en su mundo del juego y comprobará que el valor de la defensa es el mismo, para evitar la realización de trampas. En ese caso, se procederá a calcular el daño que va a recibir el jugador defensor. Para ello, comprobará primero si el valor de defensa es mayor que el de ataque (en cuyo caso no habrá daño) y si no lo es, restará el valor de la carta atacante (guardado en el objeto *GameManager* del servidor en el método *SendAttackServerRpc*) al de la carta defensora para calcular el daño total.

Después, el servidor realizará una RPC al cliente atacante con el método *GetOpponentCardClientRpc* explicado antes, al cual le enviará el nombre del objeto de la carta defensora para que este pueda instanciarla en su mundo y el jugador pueda visualizar la carta con la que su oponente se ha defendido. Esta visión será similar a la de la **Figura 66**, con la diferencia de que la carta de la esquina izquierda será la defensora y la que está en posición avanzada será la carta con la que el jugador ha realizado el ataque (además de que en el texto que informa de la situación del turno pondrá "TU TURNO").

```
[ServerRpc]
1 referencia
private void SendDefenseServerRpc(int defense, string card, ServerRpcParams serverRpcParams)
{
    Card serverCard = GameObject.Find(card).GetComponent<Card>();
    if (serverCard.defense == defense)
    {
        int totalDamage;
        if (defense >= gm.attackerDamage)
        {
            totalDamage = 0;
        }

        else
        {
            totalDamage = gm.attackerDamage - defense;
        }

        ulong defenderPlayerID = serverRpcParams.Receive.SenderClientId;

        ulong attackerPlayerID = 0;
        foreach (ulong clientID in NetworkManager.Singleton.ConnectedClientsIds)
        {
            if (clientID != defenderPlayerID)
            {
                attackerPlayerID = clientID;
                break;
            }
        }

        GetOpponentCardClientRpc(card, false, new ClientRpcParams { Send = new ClientRpcSendParams { TargetClientIds = new List<ulong> { attackerPlayerID } } });
        StartCoroutine(GameUpdate(totalDamage, defenderPlayerID, attackerPlayerID));
    }
```

Fig. 67. Implementación de la gestión de la defensa por el servidor en el prototipo de videojuego de cartas multijugador basado en turnos

El método *SendDefenseServerRpc* acabará con la ejecución de la corutina *GameUpdate* para la resolución del turno, a la que se le envían como parámetros el daño total al defensor y las IDs del jugador defensor y atacante, en ese orden. Esta corutina ejecutada en el servidor dedicado (visible en la **Figura 68**) esperará 3 segundos y modificará la salud del jugador defensor (la variable "health" de su *NetworkEntity*) restándole el daño total. También se modificarán ambas variables "turn", dándole el valor verdadero a la del jugador que acaba de defender (al cual corresponde atacar en el siguiente turno) y falso al otro jugador. Con estas modificaciones de valores realizadas en el servidor por seguridad ante las trampas, se ejecutarán automáticamente los métodos ya explicados *HealthUpdate* y *TurnUpdate*, modificando así los datos correspondientes de las interfaces de usuario y llevando a cabo las acciones necesarias para el inicio del siguiente turno.

```
IEnumerator GameUpdate(int totalDamage, ulong defenderPlayerID, ulong attackerPlayerID)
{
    yield return new WaitForSeconds(3);

    NetworkEntity playerDamaged = NetworkManager.Singleton.ConnectedClients[defenderPlayerID].PlayerObject.GetComponent<NetworkEntity>();
    playerDamaged.health.Value -= totalDamage;
    playerDamaged.turn.Value = true;

    NetworkEntity attackerPlayer = NetworkManager.Singleton.ConnectedClients[attackerPlayerID].PlayerObject.GetComponent<NetworkEntity>();
    attackerPlayer.turn.Value = false;
}
```

Fig. 68. Implementación del método para la resolución de un turno en el prototipo de videojuego de cartas multijugador basado en turnos

6. Conclusiones

Con la realización de este trabajo se ha podido observar la importancia de las técnicas y procesos de red en los videojuegos, siendo clave para el correcto funcionamiento de estos. Se han completado todos los objetivos propuestos culminando con el desarrollo de dos prototipos con géneros totalmente distintos.

Una de las principales conclusiones a las que se puede llegar con este trabajo, es la gran diferencia en la implementación de la parte red que puede llegar a haber en videojuegos en red con distintos géneros, ya que cada uno prioriza unas características de red a otras. En este caso, se ha observado como en un shooter la implementación de la parte red se enfoca en minimizar los efectos que pueda producir la latencia al necesitar tiempos de respuesta muy bajos, mientras que en un juego de cartas el efecto de la latencia no influirá en el desarrollo de las partidas, pudiendo enfocarse en realizar un mayor número de comprobaciones ante la posible realización de trampas de los jugadores.

Para poder desarrollar un videojuego en red correctamente se necesita un gran conocimiento de todos los factores que pueden influir en la experiencia de juego y sobre como implementar las distintas técnicas y procesos para evitar que estos influyan en el juego. El no tener el conocimiento necesario puede implicar que el juego consuma más recursos de los necesarios, que tenga grandes inestabilidades por latencia, que no sea capaz de evitar el uso de trampas o que los jugadores no sean capaces de encontrar una sesión de juego. Todos estos problemas implican que la experiencia de juego sea deficiente y el producto no sea apetecible para los usuarios, llevando el videojuego al fracaso dentro del sector.

Con este trabajo se pueden conocer las técnicas y procesos de red más importantes en la parte red de un videojuego y ver como se pueden llegar a implementar mediante Unity y las herramientas Photon Fusion y Netcode for GameObjects, de las que se ha podido ver sus diferencias. La conclusión principal de ambas herramientas es que mientras que Photon Fusion es la más completa en cuanto a las características que ofrece, Netcode ofrece más facilidades en la implementación de ciertos aspectos pertenecer a Unity.

Mediante este proyecto he podido aplicar conocimientos adquiridos a lo largo del grado en Ingeniería Informática relacionados con las redes y la programación. El estudio de lenguajes como Java y C++ han permitido que pueda aprender de forma rápida programar en C#, que es el lenguaje utilizado por Unity. Los conocimientos adquiridos en las asignaturas de redes han permitido que ya tuviera una base en conceptos como las distintas arquitecturas de red, las distintas capas de transporte, protocolos de transmisión de datos TCP y UDP…

Para finalizar, a nivel personal, este trabajo me ha servido para introducirme en el sector de desarrollo de videojuegos, el cual siempre me ha llamado la atención. Es un sector que me interesa desde antes de entrar en el grado en Ingeniería Informática y en el cual no había podido integrarme a lo largo del grado. Mediante este proyecto he aprendido a utilizar un motor de videojuegos y he ganado un gran conocimiento en cuanto a como están implementadas las partes red en los videojuegos al detalle. Gracias a esto, podré seguir desarrollándome en esta área mediante el desarrollo de más videojuegos para continuar en mi proceso de aprendizaje en un sector en el que podría llegar a querer dedicar mi vida laboral.

7. Trabajo futuro

La implementación de los prototipos en este trabajo de fin de grado se ha centrado principalmente en la parte red de estos, por lo que hay muchas otras partes del desarrollo de videojuegos en las que quiero mejorar estos prototipos.

Respecto al prototipo de shooter multijugador en tercera persona, tengo pensado implementar distintos modos de juego:

- Uno basado en oleadas, en las que en cada ronda se generen una cantidad X de enemigos con una dificultad determinada (la cual vaya subiendo en cada ronda) y entre los distintos jugadores conectados en red se coordinen para acabar con todos los enemigos.
- La parte actual implementada correspondería a un modo de entrenamiento.
- Un modo de juego parecido al implementado, pero con sistemas de puntuación en la partida. La idea sería también introducir un sistema de niveles para cada jugador y que al subir niveles desbloquee armas o nuevas skins. Se podría implementar en el sistema de matchmaking la búsqueda por habilidad basándose en el nivel del jugador.

También tengo intención de aprender a implementar animaciones para los personajes y skins para obtener una mejora visual sustancial en el prototipo.

En cuanto a las futuras mejoras que planteo para el prototipo de videojuego de cartas multijugador basado en turnos, tengo intención de añadir las siguientes características:

- Añadir nuevas cartas al juego y que el jugador pueda elegir las 10 cartas con las que jugar las partidas entre todas las que tenga disponibles.
- Un sistema de ligas en el que se vaya subiendo de categoría mediante victorias. El sistema de matchmaking sería modificado para unir a los jugadores a una sesión con un jugador de su liga o similar.
- Sistema de creación de torneos que puedan dar premios de puntos para liga y cartas nuevas exclusivas.

Mi intención, por tanto, es la de seguir formándome en el área del desarrollo de videojuegos mediante la mejora de los dos prototipos creados en este trabajo, aplicando funcionalidades nuevas y mejorando las actuales. También trataré de desarrollar videojuegos de otros géneros para poder seguir expandiendo mis conocimientos en el área.

Referencias

[1] egamers, «Tipos de videojuegos: Principales géneros que todo gamer debe conocer», 26 agosto 2018. [En línea]. https://www.egamers.com/2018/08/26/tipos-de-videojuegos-principales-generos-que-todo-gamer-debe-conocer/. [Último acceso: 22/09/2022].

[2] A. Sedeño Vandellós, «Videojuegos Como Dispositivos Culturales: Las Competencias Espaciales En Educación», *Comunicar,* vol. 17, nº 34, pp. 183-189, 2010. [Último acceso: 25/09/2022].

[3] S. Liu, X. Xu y M. Claypool, «A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games», *ACM Computing Surveys,* vol. 54, nº 11s, pp. 1-34, 2022. [Último acceso: 24/10/2022].

[4] C. Hansen, N. Jurgens, D. Makaroff, D. Callele y P. Dueck, «Network performance measurement framework for real-time multiplayer mobile games», *12th Annual Workshop on Network and Systems Support for Games (NetGames)*, Denver, CO, USA, 2013. [Último acceso: 29/10/2022].

[5] B. House y M. Baker, «The Future of Connected Games Unity and Google Cloud», *Unite Berlin 2018*, Berlin, Alemania, 2018. [vídeo en línea]. https://www.youtube.com/watch?v=UExWjX-S8Jw&ab_channel=Unity. [Último acceso: 23/11/2022].

[6] M. Baker y B. House, «Connected Games: Building real-time multiplayer games with Unity and Google», de *Unite Los Angeles*, Los Ángeles, USA, 2018. [vídeo en línea]. https://www.youtube.com/watch?v=CuQF7hXlVyk&t=8s&ab_channel=Unity. [Último acceso: 28/11/2022].

[7] I. Vienažindytė, «TCP vs. UDP, comparamos los dos protocolos», 13 noviembre 2019. [En línea]. https://nordvpn.com/es/blog/protocolo-tcp-udp/#:~:text=La%20principal%20diferencia%20entre%20TCP,protocolo%20UDP%20no%20lo%20es. [Último acceso: 15/12/2022].

[8] I. Méndez, «Diferencia entre TCP y UDP, comparación de los protocolos», 3 febrero 2021. [En línea]. https://linuxbasico.com/diferencia-tcp. [Último acceso: 15/12/2022].

[9] IONOS, «Remote Procedure Call: comunicación en sistemas cliente-servidor», 10 marzo 2020. [En línea]. https://www.ionos.es/digitalguide/servidores/know-how/que-es-rpc/. [Último acceso: 21/12/2022].

[10] Ajedrez.pro, «Elo», 13 julio 2018. [En línea]. https://ajedrez.pro/elo. [Último acceso: 15/01/2023].

[11] Hmong, «Sistema de clasificación Glicko». [En línea]. https://hmong.es/wiki/Mark_Glickman. [Último acceso: 16/01/2023].

[12] M. E. Glickman, «Example of the Glicko-2 system», Boston University, 22 marzo 2022. http://glicko.net/glicko/glicko2.pdf. [Último acceso: 18/01/2023].

[13] G. Jimenez Díaz , J. A. Recio García, B. Díaz Agudo y G. Flórez Puga, «Uso de competiciones y sistemas de clasificación como metodología», *Simposio-Taller JENUI 2012*, Ciudad Real, España, 2012. [Último acceso: 24/01/2023].

[14] Wikipedia, «TrueSkill», 27 febrero 2022. [En línea]. https://en.wikipedia.org/wiki/TrueSkill. [Último acceso: 20/01/2023].

[15] S. Lehtonen, «Comparative Study of Anti-cheat Methods in Video Games», Master's thesis, University of Helsinki, 2020. [Último acceso: 24/02/2023].

[16] S. Lague, «Unity Create a Game Series», 2015. [vídeo en línea]. https://www.youtube.com/playlist?list=PLFt_AvWsXl0ctd4dgE1F8g3uec4zKNRV0. [Último acceso: 30/04/2023].

[17] S. Lague, «Create a Game Source», *GitHub*. https://github.com/SebLague/Create-a-Game-Source. [Último acceso: 30/04/2023].

[18] Photon, «Fusion Introduction», 2021. https://doc.photonengine.com/fusion/current/getting-started/fusion-intro. [Último acceso: 18/06/2023].

[19] XR-AI, «Unity / PHOTON FUSION 101 Tutorials», 4 noviembre 2022. [vídeo en línea]. https://www.youtube.com/playlist?list=PL-QF7tzwZ4CeO9u1HRePeal0U4RLlf1Wo. [Último acceso: 19/05/2023].

[20] T. Waggoner, «PhotonFusion101», *GitHub*, 2022. https://github.com/xr-ai-labs/PhotonFusion101. [Último acceso: 19/05/2023].

[21] Photon, «Matchmaking API», 2021. https://doc.photonengine.com/fusion/current/manual/matchmaking. [Último acceso: 25/05/2023].

[22] Pretty Fy Games, «Tutorial series: Online multiplayer FPS with Photon Fusion & Unity», 2022. [vídeo en línea]. https://www.youtube.com/playlist?list=PLyDa4NP_nvPfHhPuumJylSj8jXyULsT1X. [Último acceso: 29/05/2023].

[23] Pretty Fly Games, «Multiplayer FPS Unity & Photon Fusion tutorial project», 28 abril 2022. [En línea]. https://www.patreon.com/posts/multiplayer-fps-65725126. [Último acceso: 13/05/2023].

[24] Photon, «NetworkTransform Class Reference», 2021. https://doc-api.photonengine.com/en/fusion/current/class_fusion_1_1_network_transform.html. [Último acceso: 15/05/2023].

[25] Pretty Fly Games, «Tutorial: Online multiplayer FPS Unity & Photon Fusion EP3 (shooting + health)», 19 junio 2022. [En línea]. https://www.patreon.com/posts/tutorial-online-67984961. [Último acceso: 20/05/2023].

[26] Pretty Fly Games, «Tutorial: Online multiplayer FPS Unity & Photon Fusion EP7 (Lobby Session Browser)», 23 noviembre 2022. [En línea]. https://www.patreon.com/posts/tutorial-online-75034121. [Último acceso: 29/05/2023].

[27] Blackthornprod, «How to make a CARD GAME - Unity Tutorial 2022», 30 enero 2022. [vídeo en línea]. https://www.youtube.com/watch?v=C5bnWShD6ng&t=44s&ab_channel=Blackthornprod. [Último acceso: 02/06/2023].

[28] Code Monkey, «COMPLETE Unity Multiplayer Tutorial (Netcode for Game Objects)», 26 septiembre 2022. [vídeo en línea]. https://www.youtube.com/watch?v=3yuBOB3VrCk&ab_channel=CodeMonkey. [Último acceso: 05/06/2023].

[29] Tarodev, «How to setup Global Matchmaking for Unity», 23 julio 2022. [vídeo en línea]. https://www.youtube.com/watch?v=fdkvm21Y0xE&ab_channel=Tarodev. [Último acceso: 08/06/2023].

[30] M. Spencer, «SimpleMatchmaking.cs», *GitHub,* 2023. https://gist.github.com/Matthew-J-Spencer/a5ab1fb5a50465e300ea39d7cde85006. [Último acceso: 08/06/2023].

[31] Unity, «Unity Relay Documentation», 2022. https://docs.unity.com/relay/en/manual. [Último acceso: 25/06/2023].

[32] Unity, «Unity Lobby Service Documentation», 2022. https://docs.unity.com/lobby/en/manual. [Último acceso: 25/06/2023].

[33] I. Asensio, «¿Qué es Unity y para qué sirve?», 6 noviembre 2019. [En línea]. https://www.masterd.es/blog/que-es-unity-3d-tutorial. [Último acceso: 16/06/2023].

[34] Starloop Studios, « Unreal vs. Unity 3D: Choosing the Best Engine for Your Game», febrero 2023. [En línea]. https://starloopstudios.com/unreal-vs-unity-3d-choosing-the-best-engine-for-your-game/. [Último acceso: 16/06/2023].

[35] Microsoft, «Paseo por el lenguaje C#», 15 febrero 2023. https://learn.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/. [Último acceso: 18/06/2023].

[36] Photon, «INetworkRunnerCallbacks Interface Reference», 2021. https://doc-api.photonengine.com/en/fusion/current/interface_fusion_1_1_i_network_runner_callbacks.html. [Último acceso: 18/06/2023].

[37] Unity, «Dedicated Game Server (DGS)», 3 febrero 2023. https://docs-multiplayer.unity3d.com/netcode/current/reference/dedicated-server/. [Último acceso: 19/06/2023].

[38] Photon, « Network Input», 2021. https://doc.photonengine.com/fusion/current/manual/network-input. [Último acceso: 20/06/2023].

[39] Photon, «Lag Compensation», 2021. https://doc.photonengine.com/fusion/current/manual/lag-compensation. [Último acceso: 21/06/2023].

Anexo I: Presupuesto

En este anexo del documento se va a detallar el presupuesto necesario para llevar a cabo este proyecto, incluyendo los materiales de hardware y software utilizados y la mano de obra. En la **Tabla 3** se puede ver el coste de cada uno de los materiales de software y hardware utilizados para realizar el trabajo. El cálculo del precio del portátil y del ordenador de sobremesa se han hecho en base a los meses de uso para que sea un precio más justo, tomando como referencia que cada dispositivo tendrá 3 años de vida útil (36 meses). El precio de compra del portátil es de 549,99 € y del ordenador de sobremesa de 769,99 €, por lo que se ha dividido ese precio entre 36 y se ha multiplicado por los meses de uso para sacar el coste imputable del uso de cada dispositivo.

TABLA III
Coste de los materiales de software y hardware utilizados en el proyecto

Material	Precio
Portátil Acer Nitro 5 AN515-55 (9 meses de uso): - CPU Intel Core i5-10300h - 16 GB de memoria RAM - GPU NVIDIA GeForce GTX 1650	137,50 €
Ordenador de sobremesa (2 meses de uso): - CPU Intel Xeon E5-2660 v3 - 32 GB de memoria RAM - GPU NVIDIA GeForce GTX 1080	42,78 €
Monitor HP 27Q 27" LED QuadHD	179,99 €
2x Sistema operativo Microsoft Windows 10 Pro	238,00 €
Teclado	29,99 €
Ratón	15,99€
Unity	0 €
Visual Studio Community 2022	0 €
Photon Fusion	0 €
Netcode for GameObjects	0 €
Coste total de los materiales	644,25 €

Como se observa en la **Tabla 3**, el software utilizado para el proyecto ha sido gratuito, a excepción de las licencias del sistema operativo Windows 10 que se han utilizado. El coste total de los materiales utilizados asciende a 644,25 €.

En la **Tabla 4** se puede ver el coste de la mano de obra en el proyecto desglosado por tareas. Dependiendo de la dificultad de la tarea el coste por hora será uno u otro, al igual que en una empresa dependiendo del rango del puesto del trabajador el coste por hora es uno u otro. El coste total de mano de obra en base a la estimación de horas en el proyecto es de 16.040 €.

TABLA IV
Coste de la mano de obra en el proyecto

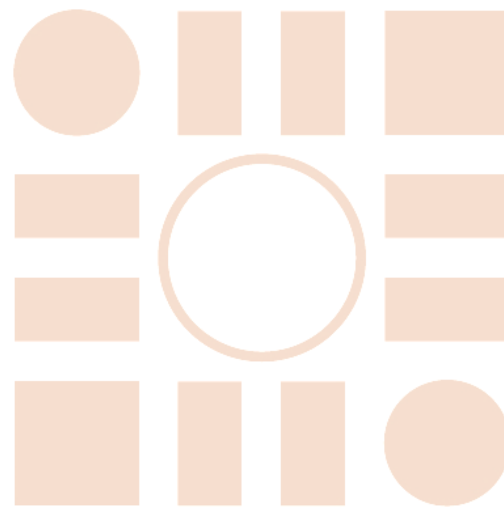
Tarea	Número de horas	Coste hora	Precio
Recopilación de información	100	22 €	2200 €
Estudio de la información recopilada	120	26€	3120 €
Diseño de los prototipos	100	46 €	4600 €
Programación de los prototipos	180	34 €	6120 €
Coste total de la mano de obra	500		16.040 €

Se ha realizado también una estimación de gastos generales en el proyecto correspondientes a los gastos de luz y mantenimiento, los cuales se estiman en un 12% de los gastos de materiales y de mano de obra, lo cual supone una cantidad de 2002,11 €. Además, el beneficio industrial del proyecto se ha estimado en un 15% de la suma total del coste de materiales, mano de obra y gastos generales (18.686,36 €), correspondiendo a un beneficio de 2802,95€. Realizado la suma de los cuatro costes, el presupuesto total del proyecto se estima en 21.489,31 €. Todo esto se puede ver en la **Tabla 5**, en la cual se encuentra desglosado el presupuesto total del proyecto.

TABLA V
Presupuesto total del proyecto

Tipo de coste	Precio
Coste total de los materiales de software y hardware	644,25 €
Coste total de la mano de obra	16.040 €
Gastos generales (luz y mantenimiento): 12% del coste total de los materiales y la mano de obra	2002,11 €
Beneficio industrial: 15% del coste total de los materiales, la mano de obra y los gastos generales del proyecto	2802,95 €
Presupuesto total del proyecto	21.489,31 €

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá