UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un videojuego multijugador en red con Unity 3D

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Colom Colom, Joan

Tutor: Agustí Melchor, Manuel

2020 / 2021

Resumen

El presente trabajo recoge el desarrollo de un juego multijugador y en red sobre Unity, con el propósito de determinar cuáles son los pasos necesarios para convertir un juego local en un juego en red. Abordamos dicho desarrollo con el objetivo de que nuestro trabajo sirva como referencia a futuros desarrolladores de videojuegos multijugador acerca de las posibilidades y técnicas a utilizar en un campo en pleno cambio.

Primero estudiaremos las modificaciones que ha sufrido a lo largo de los últimos años la solución de Unity, analizando la obsolescencia de UNet y cómo esta ha llevado a Unity a apostar por la solución de MLAPI. A continuación, presentaremos un juego de carrera infinita 2D que incluirá todos los aspectos básicos de un juego moderno. A partir de este ejemplo, procederemos a detallar el proceso de adaptación del mismo a una versión multijugador en línea basada en UNet.

Uno de los propósitos de un juego multijugador es conectar a las personas en un entorno de diversión. Con esta idea en mente y reforzada por la actual situación global, nos plantearemos expandir la versión multijugador básica para la incorporación de funcionalidades que aprovechen la conectividad para enriquecer la experiencia, como la transmisión tanto de vídeo como de audio para explorar las capacidades de UNet en este ámbito y permitir a los jugadores una mayor capacidad de comunicación.

Finalmente, procederemos a la transición del proyecto a MLAPI. De esta manera, exploraremos la perspectiva de futuro para los juegos multijugador en Unity y las capacidades del nuevo paquete de desarrollo.

Gracias a estos desarrollos, podremos estudiar las diferentes técnicas y funciones que permiten tanto UNet como MLAPI para abordar aspectos como la lógica de juego, la sincronización de jugadores en tiempo real y la interacción entre jugadores mediante chat de texto, audio y vídeo. Para terminar, obtendremos los resultados de los desarrollos llevados a cabo y haremos hincapié en los pasos a seguir en el desarrollo de juegos multijugador en línea.

Palabras clave: Unity, UNet, MLAPI, red, videojuego, multijugador, comunicación.

Abstract

This work exposes the development of a multiplayer networked game in Unity, with the purpose of determining which are the required steps to convert a local game into a networked game. We approach this development with the aim of offering with our work a reference to future multiplayer videogame developers about the possibilities and techniques to be used in a field in process of changing.

First, we will study the changes that Unity's networking solution has undergone in recent years, analyzing the depreciation of UNet and how it has led Unity to bet on the MLAPI solution. Next, we will present a 2D infinite run game that will include all the basics of a modern game. From this example, we will proceed to detail the process of adapting it to an online multiplayer version based on UNet.

One of the purposes of a multiplayer game is to connect people in a fun and enjoyable environment. With this idea in mind and reinforced by the current global situation, we will consider expanding the basic multiplayer version to incorporate functionalities that take advantage of connectivity to enrich the experience, such as the transmission of both video and audio to explore the capabilities of UNet in this area and give players better communication tools.

Finally, we will proceed to migrate the project to MLAPI. In this way, we will explore the future for multiplayer games in Unity and the capabilities of the new API.

Thanks to these developments, we will be able to study the different techniques and functions that both UNet and MLAPI allow to address aspects such as game logic, synchronization of the players in real time and allowing the interaction between players through text chat, audio and video. To conclude, we will expose the results obtained from the developments carried out and we will remark the steps to follow in the development of networked multiplayer games.

Keywords : Unity, UNet, MLAPI, network, videogame, multiplayer, communication.

Resum

L’actual treball recull el desenvolupament d'un joc multijugador i en xarxa sobre Unity, amb el propòsit de determinar quins són els passos necessaris per convertir un joc local en un joc en xarxa. Abordem aquest desenvolupament amb l'objectiu que la nostra feina servisca com a referència per a futurs desenvolupadors de videojocs multijugador sobre les possibilitats i tècniques a utilitzar en un camp en ple canvi.

Primer estudiarem les modificacions que ha sofrit al llarg dels últims anys la solució d'Unity, analitzant l’obsolescència d’UNet i com aquesta ha portat a Unity a apostar per la solució de MLAPI. A continuació, presentarem un joc de carrera infinita 2D que inclourà tots els aspectes bàsics d'un joc modern. A partir d'aquest exemple, procedirem a detallar el procés d'adaptació del mateix a una versió multijugador en línia basada en UNet.

Un dels propòsits d'un joc multijugador és connectar a les persones en un entorn de diversió. Amb aquesta idea en ment i reforçada per l'actual situació global, ens plantejarem expandir la versió multijugador bàsica per a la incorporació de funcionalitats que aprofiten la connectivitat per enriquir l'experiència, com la transmissió tant de vídeo com d'àudio per explorar les capacitats d’UNet en aquest àmbit i permetre als jugadors una major capacitat de comunicació.

Finalment, procedirem a la transició del projecte a MLAPI. D'aquesta manera, explorarem la perspectiva de futur per als jocs multijugador en Unity i les capacitats del nou paquet de desenvolupament.

Gràcies a aquests desenvolupaments, podrem estudiar les diferents tècniques i funcions que permeten tant UNet com MLAPI per abordar aspectes com la lògica de joc, la sincronització de jugadors en temps real i la interacció entre jugadors mitjançant xat de text, àudio i vídeo. Per acabar, obtindrem els resultats dels desenvolupaments duts a terme i farem èmfasi en els passos a seguir en el desenvolupament de jocs multijugador en línia.

Palabras clave: Unity, UNet, MLAPI, xarxa, videojoc, multijugador, comunicació.

Índice de contenidos

Índice de figuras

Índice de tablas

1.Introducción

A lo largo de este trabajo vamos a ver, a través de la perspectiva de un videojuego, cómo se puede abordar desde Unity, también conocido a veces como Unity 3D, el despliegue de una aplicación interactiva entre usuarios físicamente separados. Así, haremos uso de la red para conseguir que exista un flujo de información entre los jugadores en diferentes formatos, como son la imagen, el texto o el sonido. El diseño de unas mecánicas de juego simples nos permitirá centrar nuestra atención en la definición e implementación de funcionalidad en línea y los sistemas de comunicación entre los usuarios.

1.1.Motivación

Los videojuegos multijugador en red son un tópico de gran actualidad y, ahora más que nunca, son una forma muy importante de conectar a personas independientemente de dónde se encuentren. En estos tiempos de distanciamiento social, los sistemas interactivos en línea como los videojuegos o los sistemas de videoconferencia han sido de vital importancia y apoyo para multitud de personas que se han encontrando aisladas, siendo una vía de escape de la realidad, de reunión con los seres queridos y de encuentro para el trabajo o la docencia en remoto. Este trabajo de fin de grado tiene como motivación el estudio de las tecnologías que permiten la unión de personas en entornos virtuales que facilitan la interacción con los demás en un entorno lúdico, es decir, el estudio de los videojuegos que permiten el acercamiento e interacción entre personas.

Por otro lado, el campo de los juegos en red basados en Unity ha sufrido cambios en los últimos años que han resultado en una situación incierta y con una documentación a veces escasa o incompleta. Por ello, creemos que esta situación requiere un estudio e investigación apropiados para aclarar cuáles son las alternativas de desarrollo y el futuro de los juegos en red basados en Unity. Queremos así con nuestro trabajo proporcionar un apoyo y referencia para futuros desarrolladores de juegos multijugador en Unity, aclarando el camino que se puede seguir.

1.2.Objetivos

Los objetivos que nos hemos marcado con este proyecto son:

- Determinar cuáles son los pasos necesarios para la conversión de un videojuego local para un solo jugador desarrollado en Unity en un videojuego multijugador en red.
- Identificar la situación actual en la que se encuentra la solución multijugador ofrecida por Unity, analizando cómo se ha llegado a la misma.
- Explorar el futuro de los juegos multijugador en Unity.
- Determinar las capacidades que ofrece Unity para el desarrollo de videojuegos en red.
- Determinar las capacidades que ofrece Unity para el desarrollo de sistemas que faciliten la comunicación e interacción remota entre personas.

1.3. Estructura

El presente documento se estructura en un total de ocho capítulos. A lo largo de los mismos se introducirá el tópico principal del trabajo, se analizará su estado del arte, se presentará el desarrollo del proyecto y finalmente analizaremos los resultados obtenidos, contraponiéndolos con los objetivos y detallando posibles trabajos futuros a desarrollar. De esta manera:

- El primer capítulo introduce el tópico a tratar. Se establecen además los objetivos del proyecto y la motivación para su desarrollo.
- El segundo capítulo presenta la situación actual en el ámbito del desarrollo de juegos multijugador en Unity y cuáles son los cambios que ha ido sufriendo la solución multijugador de Unity. Exploraremos también soluciones de terceros compatibles con el motor de Unity.
- El tercer capítulo se dedica a la introducción de los conceptos necesarios para el trabajo como las llamadas remotas y la tipología de los juegos multijugador.
- El cuarto capítulo expone la metodología y herramientas utilizadas durante el desarrollo del proyecto.
- El quinto capítulo procederá a detallar el desarrollo del proyecto. Dicho desarrollo pasará por la propuesta de un sencillo juego local en el que se cubren aspectos comunes y esenciales a la mayoría de videojuegos. A partir de este ejemplo, se analizarán y diseñarán las características de su versión multijugador, detallando finalmente cómo implementar dicho diseño mediante la solución tradicional multijugador de Unity. Finalmente, se analizará el futuro del desarrollo de juegos multijugador mediante la transición de nuestro videojuego multijugador a la solución de desarrollo más reciente, analizando las diferencias y compatibilidad.
- El sexto capítulo se destina al análisis de los resultados obtenidos del desarrollo.
- El séptimo capítulo extraerá las conclusiones que podemos obtener a partir de los resultados, de acuerdo a los objetivos propuestos.
- Finalmente, el octavo capítulo presenta los posibles caminos por los que el desarrollo del proyecto podría proseguir.

2.Estado de la cuestión

El desarrollo de videojuegos multijugador haciendo uso de Unity se encuentra en una situación compleja y poco clara en primera instancia. A lo largo de este apartado recopilaremos los diferentes eventos y cambios que ha sufrido la solución para juegos multijugador en red de Unity en los últimos años con el fin de esclarecer la situación actual.

En 2014, Unity anunció la incorporación de UNet [3] a la familia de APIs de Unity para facilitar el desarrollo de juegos multijugador en red. UNet estaba formado por varios componentes [44] que proporcionaban las herramientas necesarias para el desarrollo de este tipo de juegos. Los principales son:

- Una capa de transporte basada en UDP.
- *Low Level API* (LLAPI), interfaz de programación a bajo nivel que permite el control de los sockets.
- *High Level API* (HLAPI), interfaz de programación de alto nivel que permite administrar la red a nivel de clientes y servidores.
- Un servicio *Matchmaker*, basado en los servicios de nube multijugador de Unity.
- Servicios de autenticación que daban el soporte básico necesario para validar sesiones, aunque no proporcionaban un sistema de identificación robusto.

No obstante, en 2018, Unity anunció el fin del soporte para UNet [13] en favor de una nueva alternativa a la que denominaban *Unity Connected Games* [26]. *Connected Games* tenía como objetivo mejorar el soporte y la eficiencia necesaria para el desarrollo y mantenimiento de juegos en red a gran escala, motivo por el cual se planteaba como una colaboración con Google para aprender de su experiencia en el uso de servidores. Por tanto, *Connected Games* era un conjunto de nuevas incorporaciones formado por:

- Una nueva API de red que prometía tener como objetivo proporcionar una mayor eficiencia al ser compatible con el nuevo paradigma DOTS [12] de Unity, el cual se basa en una programación orientada a los datos. Dicha nueva API también implicaba la realización por parte de Unity de una nueva pila de red al completo, comenzando por una nueva capa de transporte basada en UDP.
- Un mayor soporte para juegos basados en servidores dedicados, proporcionando mayor seguridad, consistencia y escalabilidad. Esto se prometía cubrir gracias a la adquisición e incorporación en Unity de *Multiplay* [58], la cual ya daba soporte en su momento a grandes juegos en red.
- Un nuevo servicio de servidores mediante la integración con *Google Cloud*, con el objetivo de reducir el coste de mantener un juego en servidores dedicados.
- Un nuevo servicio de *Matchmaking* llamado *Open Match*. Este servicio se planteaba como un proyecto de código abierto realizado en colaboración con Google que pretendía reemplazar al sistema *Matchmaking* de UNet, integrándose con las tecnologías de *Multiplay* para una mayor escalabilidad. Dicho servicio también

prometía mayores capacidades de personalización para crear una lógica de partida propia mediante C#.

De esta forma, Unity plantea la actualización de UNet como una transición progresiva hacia el nuevo *Connected Games*. Según esta transición que podemos ver en la Figura 1, principios del 2021 se establecía como el fin de la HLAPI de UNet y su reemplazo por parte de *Connected Games*.

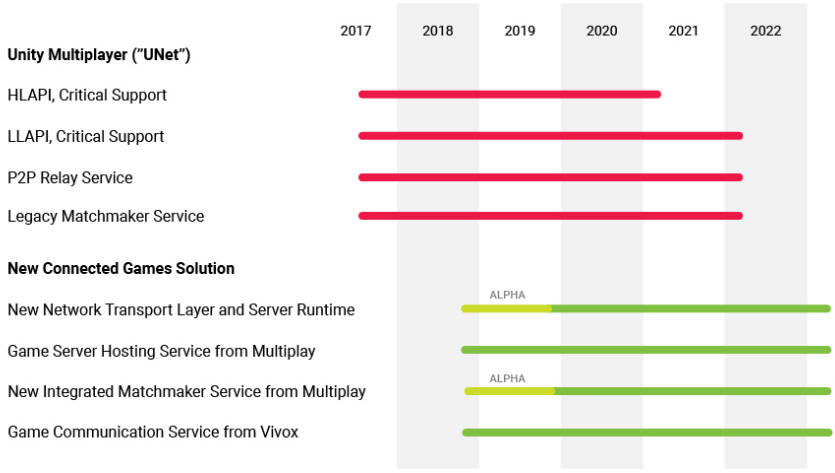

Figura 1: proceso de transición entre UNet y *Connected Games* [13].

Sin embargo, en 2019 se actualiza el plan de extinción de UNet para extender el soporte crítico del LLAPI. Así, en este año, encontramos solamente un artículo de Unity en el que se vuelva a hablar de la nueva API de juego en red y el último artículo de Unity en el que oímos hablar de *Connected Games* [28]. En dicho artículo, se exponen las tres alternativas para el desarrollo de videojuegos en red: el ya obsoleto UNet, el nuevo API para juegos en red basado en DOTS y la API de transporte de bajo nivel, analizando cuál es la mejor alternativa según las características del proyecto a desarrollar. En este artículo también se especifica el objetivo de tener la nueva API de juego en red, a la se refieren con el nombre de *NetCode* (término que ya persistirá hasta la fecha), preparada y lista para producción a principios del año 2021. Como podemos ver, el lanzamiento de la nueva API comienza a retrasarse en este punto con respecto a la planificación original de la Figura 1.

En 2020 vuelve a incrementar la actividad informativa de Unity con respecto a la API de red. Se presenta el mapa de ruta hacia el 2021 [40] y, en él, se explica que se están centrando en proporcionar un abanico completo de soluciones para los diferentes tipos clave de juegos multijugador, por lo que se comprometen a extender el soporte de la nueva API más allá de DOTS para cubrir también el actual sistema basado en *GameObjects*.

Este mismo año Unity publica un artículo clarificando las diferentes alternativas posibles para el desarrollo de juegos multijugador en red [5], puntuando las mismas según su soporte, facilidad de uso, eficiencia, escalabilidad, características y coste. Dichas alternativas son:

- MLAPI [59]: se recomienda como la mejor alternativa para el desarrollo debido a su abanico de características tanto de alto como de bajo nivel, su activa comunidad y su

código abierto. Cabe destacar que en el momento en el que se publica este artículo MLAPI aún no formaba parte de la familia de Unity.

- *DarkRift 2* [11]: una API de bajo nivel que asume una topología con servidores dedicados. Esta alternativa se recomienda para desarrolladores que busquen una alternativa de alta eficiencia y escalabilidad, estando dispuestos a diseñar sus propias capas de medio y alto nivel sobre ella. Esta alternativa implica un pago para poder acceder a todas las características.

- *Photon PUN* [34]: solución basada en una topología P2P directa. Unity recalca el inconveniente que supone este tipo de topología en cuanto a su escalabilidad y la carencia de servidor de autoridad para prevenir engaños. Por tanto, esta alternativa se recomienda para juegos cooperativos de pequeña escala que impliquen pocos jugadores simultáneos. Esta alternativa impone un pago para más de 20 jugadores simultáneos.

- *Photon Quantum v2* [4]: cubre el apartado de simulación en un motor de videojuegos orientado a juegos deterministas, haciendo uso de un Sistemas de Entidades y Componentes (ECS) y sistemas multihilo. El uso de esta alternativa implica utilizar las APIs de Photon para físicas, matemáticas y búsqueda de caminos, entre otras, en lugar de las proporcionadas por Unity. Esta solución también implica un coste y tiene el inconveniente de basar altamente su funcionamiento en el hardware cliente para llevar a cabo la simulación de todos los jugadores.

- *Mirror* [24]: otra API de código abierto, la cual consiste en un *fork* de la HLAPI de UNet que ha sido evolucionado y mantenido por la comunidad. Sus ventajas recaen en la alta similitud y compatibilidad con HLAPI, siendo fácil de utilizar. Además, sus desarrolladores se han encargado de corregir y mejorar los errores detectados en UNet. *Mirror* se encuentra disponible en la *Asset Store* de Unity de forma gratuita y tiene una amplia documentación.

	Stability/ support	Ease-of-use	Perfor-mance	Scalability	Feature breadth	Cost*	Customers recommend for
MLAPI	★★★★★	★★★★☆	★★★★★	★★★★☆	★★★★★	Free	Most client-server games for up to ~64 players that want a stable breadth of mid-level features
DarkRift 2	★★★★★	★★★☆☆	★★★★★	★★★★★	★★★☆☆	$100 for source	Games with high perf/ scale requirements that want to build on a stable LL layer
Photon PUN	★★★★☆	★★★★★	★★★☆☆	★☆☆☆☆	★★★★☆	$0.30/PCU	Simple and small (2–8 players) mesh-topology games
Photon Quantum 2.0	★★★★☆	★★★★☆	★★★★★	★★☆☆☆	★★★★☆	$1000/mo + $0.50/PCU	Games desiring deterministic roll-back, like MOBA games, for up to 32 players
Mirror	★★★★★	★★★★★	★★★☆☆	★★★★☆	★★★★☆	Free	Stable and proven client-server solution, loved best for its community and ease-of-use

* Note that Photon pricing provides access to the networking libraries and services, whereas other solutions are standalone networking libraries, and the cost of services is separate.

Tabla 1: valoración de Unity de las diferentes alternativas para el desarrollo de juegos en red [5].

Como se puede ver, en este artículo se deja de lado la solución DOTS de *NetCode*, según indica la actualización del artículo, para evitar confusiones. De esta lista podemos ver cómo en el momento de su elaboración Unity aún está desprovista de una solución propia para dar soporte a los juegos en red basados en *GameObjects* que reemplace la HLAPI de UNet.

Finalmente para terminar el año, en diciembre de 2020 Unity anuncia la incorporación a su familia de APIs del proyecto de MLAPI [1]. Gracias a esta API, Unity incorporaba el prometido soporte para juegos en red basados en *GameObjects*. Con el fin de ofrecer mayor claridad, hemos ordenado y resumido los eventos desde la declaración del fin de soporte para UNet hasta la incorporación de MLAPI en el esquema de la Figura 2.

Para entender completamente las decisiones de Unity hasta este momento, hay que tener en cuenta que en 2021 se terminaba el soporte crítico para el HLAPI y Unity seguía desprovisto de una alternativa propia ya que su reemplazo, *NetCode*, seguía en desarrollo. Por tanto, en vistas de que el modelo DOTS estaba aún lejos de su implantación y que los usuarios demandaban una clara alternativa para el desarrollo basado en *GameObjects*, Unity tuvo que tratar de cubrir el vacío existente. La adquisición de MLAPI les permitía tratar de cubrir este vacío con mayor rapidez [55]. Además, como hemos visto en la Tabla 1, MLAPI era la alternativa mejor valorada por Unity. De está manera MLAPI se convierte en el objetivo principal en el apartado de red de Unity para el 2021 [27].

Actualmente en junio del 2021, la API de *NetCode* orientada exclusivamente a DOTS [56] se encuentra en su versión 0.6.0 en estado alpha. Por otro lado, MLAPI [16] se encuentra en su versión 0.1.0 y su lanzamiento se prevé para la versión de Unity 2021.1. Si entramos en la página de Unity y navegamos hasta su sección dedicada al juego en red [29], veremos que la API que principalmente se nos indica para el desarrollo de videojuegos en red es MLAPI, y su documentación ya contiene indicaciones para llevar a cabo la adaptación desde UNet. La API de *NetCode* se reserva para ser mencionada junto con el resto de tecnologías DOTS. No obstante, a fecha de hoy, la integración de MLAPI en Unity aún está en desarrollo y no ha sido lanzada oficialmente, siendo necesario para su uso añadir el paquete desde su repositorio de GitHub [59]. La otra alternativa para poder elaborar juegos multijugador basados en *GameObjects* sigue siendo UNet, aún funcional en la versión 2020.3.10f1 de Unity, aunque hay que tener en cuenta que está marcada como obsoleta desde 2018 y que el periodo de soporte crítico termina este año.

Con respecto a las colaboraciones de Unity con Google para la integración de Google Cloud y el desarrollo del proyecto *Open Match*, no se ha proporcionado ninguna información desde 2018. En cuanto a *Multiplay*, a día de hoy es una alternativa poco clara, ya que se insta a los interesados a contactar para saber más al respecto de la misma y se propone como una alternativa para proyectos de gran escala [2].

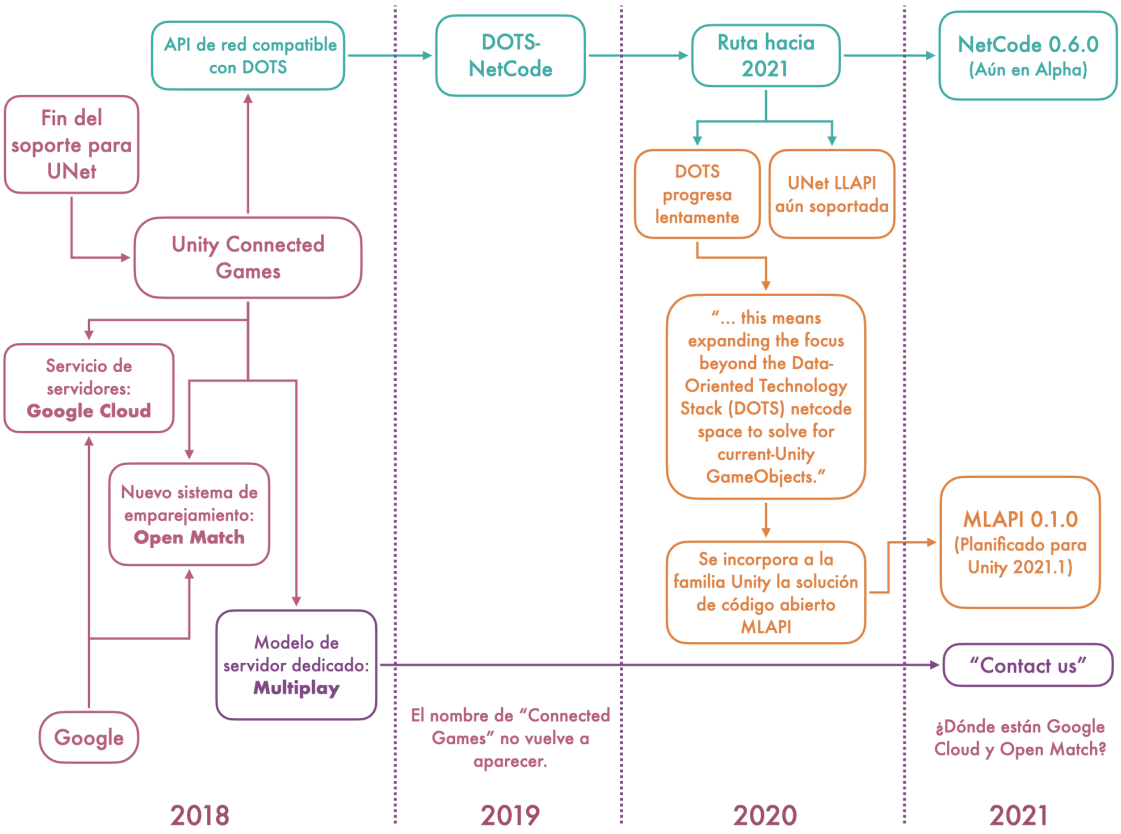

Figura 2: esquema de la transición de UNet a *NetCode*.

3.Fundamentos teóricos

En este capítulo vamos a centrar nuestra atención en los conceptos teóricos relativos al mundo de los videojuegos multijugador en línea, que es nuestro tema principal. Para ello, primero estudiaremos los diferentes tipos de videojuegos multijugador que existen y sus requisitos. A continuación, veremos las diferentes arquitecturas que dan soporte a los juegos en línea y, finalmente, abordaremos el funcionamiento de uno de los mecanismos más empleados para la sincronización en este tipo de aplicaciones interactivas.

3.1.Tipología de videojuegos multijugador

Los juegos multijugador no forman una categoría atómica de videojuegos, sino que existen diferentes niveles de conectividad que dan lugar a diversos tipos de juegos. Cada uno de estos tipos o "niveles" tiene unas necesidades técnicas diferentes. Por tanto, es importante conocer estos tipos antes de comenzar el desarrollo de cualquier videojuego multijugador para poder identificar el nivel al que pertenece un proyecto y poder determinar nuestras necesidades. A continuación detallaremos una clasificación basada en la propuesta por Unity durante el evento Unite Berlin 2018 [43], la cual podemos ver en la Figura 3.

Antes de empezar, cabe mencionar un tipo de juegos multijugador no incluido en esta clasificación y que son los juegos multijugador locales. Estos son aquellos juegos que permiten, dentro de una misma máquina, jugar de forma simultánea a múltiples usuarios. Este tipo de juegos son la base y precursor para los juegos multijugador en red, pero sus requisitos son muy similares a los de los juegos de un solo jugador ya que no requieren una transmisión a través de la red.

Así, en primer lugar, tenemos la categoría L1 [43], formada por todos aquellos videojuegos que involucran principalmente a un solo jugador, pero gracias a la conexión a través de la red permiten características como tablas de clasificación y desafíos. El núcleo principal de estos juegos es local, pero extienden la experiencia al incorporar la existencia de otros jugadores en forma de clasificaciones o eventos en línea comunes. Implican el menor nivel de interacción, por lo que sus requisitos de conectividad son menores y no tienen requisitos de tiempo real por lo general.

En segundo lugar, tenemos la categoría L2 [43], formada por aquellos juegos multijugador basados en turnos. En este tipo de juegos sí que existe una interacción directa entre los jugadores, lo que implica la presencia de un sistema de comunicación entre los dispositivos que permita sincronizar las acciones realizadas en cada turno. No obstante, la existencia de turnos ofrece una mayor tolerancia a los retrasos en las comunicaciones permitiendo el uso de tecnologías asíncronas.

En tercer lugar tenemos los juegos multijugador en tiempo real en la categoría L3 [43], que son aquellos en los que los jugadores interactuarán en tiempo real dentro de partidas multijugador. Este tipo de juegos requieren una sincronización continua del estado de los jugadores y, en general, mecanismos de predicción para tratar de ofrecer una experiencia en tiempo real. La necesidad de estos mecanismos reside en que, aun en las mejores condiciones posibles de red, las acciones no son instantáneas y tienen asociadas una latencia debido a la necesidad de transmisión de un dispositivo a otro.

Esta latencia causa desincronización durante una secuencia de cuadros, la cual se mitiga gracias a la utilización de la predicción de la acción del jugador durante estos cuadros.

Finalmente, tenemos la categoría L4 [43], que está compuesta por los juegos multijugador en tiempo real que cuentan con un mundo persistente. En lugar de organizar el juego como partidas que tienen lugar de forma temporal y se desalojan al terminar como los juegos de L3, son aquellos juegos que alojan un mundo persistente al que los jugadores esperan poderse conectar en cualquier momento y que las modificaciones que realicen sobre su avatar o sobre el mundo serán permanentes. Este tipo de juegos son los que implican una mayor dificultad técnica, ya que es necesario disponer de una infraestructura completa de servidores y bases de datos que garanticen la persistencia y disponibilidad del sistema. Esto nos remite al conocido teorema CAP [38], el cual establece la imposibilidad del diseño de un sistema distribuido que proporcione en todo momento y al mismo tiempo consistencia, disponibilidad y tolerancia a particiones de la topología.

Este sistema de clasificación de videojuegos en red es un sistema jerárquico, por el que las necesidades y dificultades presentes en un nivel también están presentes en el nivel superior. De esta manera, por ejemplo, L4 hereda las dificultades de los sistemas necesarios para L3, L2 y L1. Además, cuanto mayor sea el nivel, mayor será la interactividad y conectividad entre los jugadores, ofreciendo mayores oportunidades de comunicación.

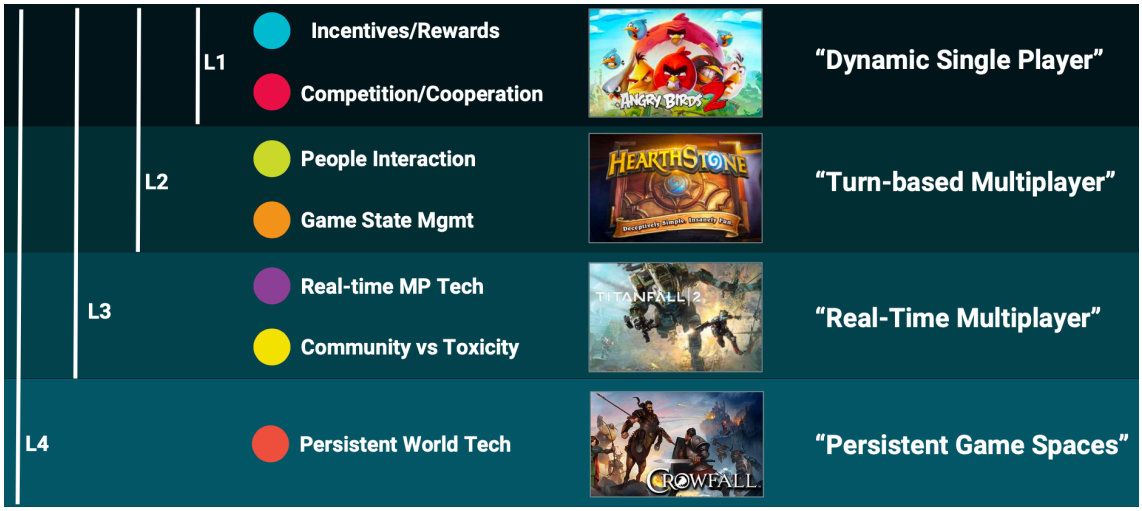

Figura 3: jerarquía de clasificación para juegos multijugador extraída de Unite Berlín 2018 [43].

3.2. Topología de redes orientadas a videojuegos

A la hora de desarrollar juegos multijugador en red, existen diferentes alternativas respecto a la topología que podemos emplear. La elección de una topología u otra dependerá de las necesidades de nuestro juego como el tipo de conectividad que ofrece, sus características o el nivel de seguridad y prevención de trampas que proporciona, entre otros.

Una alternativa es el empleo de una topología de red entre pares (*peer-to-peer* o P2P) [8][33] en la cual cada máquina se conecta de igual forma a todas las demás. Este

modelo tiene la ventaja de no recaer el juego sobre un solo servidor, de forma que si uno de los pares cae, el juego podrá continuar ya que son los propios clientes los que se encargan de la ejecución de la partida, llevando a cabo la sincronización y el intercambio de información. No obstante, esta topología tiene el inconveniente de generar una gran carga de trabajo sobre los clientes, que deberán ser capaces de administrar conexiones n-arias y mantener la sincronización con los demás. Además, esta topología dificulta las tareas de prevención de trampas y seguridad al no disponer de un servidor autoritativo.

Otra alternativa posible es el uso de un modelo cliente-servidor host, por el cual uno de los clientes toma el papel de servidor y todos los demás se conectan a él [8]. Este tipo de topología centraliza la conexión, permitiendo que cada cliente únicamente tenga que mantener la conexión con el host y reduciendo así la carga de trabajo de los clientes. No obstante, el cliente host tendrá una carga de trabajo elevada, administrando las conexiones y sincronización entre los demás jugadores. Esta topología permitiría emplear el host como mecanismo para la prevención de trampas por parte de los clientes al recibir todas las solicitudes, aunque al actuar también como cliente, cabría el riesgo de que fuera el propio host el causante de las trampas. Además, si el host se desconecta, el juego se interrumpe, siendo necesario administrar un proceso de elección de un nuevo host y reconexión al mismo de todos los clientes si queremos poder recuperar la partida en curso.

Una variante del modelo cliente-servidor host consiste en hacer uso de un servidor de reenvío entre los clientes y el host que se encargue de propagar los mensajes recibidos al receptor correspondiente [8]. De esta manera, se reduce el número de conexiones que debe mantener abiertas el host, teniendo que conectarse únicamente al servidor de reenvío. Sin embargo, esta topología puede causar un aumento de la latencia ya que cualquier transmisión de datos debe pasar primero por el servidor de reenvío antes de poder llegar a su destino, en lugar de seguir un camino directo desde el emisor al receptor. Una ventaja de esta topología es que el host únicamente conoce la IP del servidor de reenvío, no quedando al descubierto las direcciones IP de los clientes.

Finalmente, tenemos la topología basada en servidor dedicado, de tal forma que todos los jugadores actúan como clientes conectados a dicho servidor [8]. Este tipo de topología es la que presenta una mayor escalabilidad y control, ya que se pueden determinar las características hardware del servidor que mantendrá las conexiones y hará posible el juego. Además, esta topología permite una mayor seguridad y robustez ante posibles trampas, ya que el servidor puede encargarse de registrar acciones sospechosas y bloquearlas, impidiendo que un cliente jaqueado afecte la experiencia de juego de los demás jugadores. No obstante, esta es la topología más compleja ya que requiere el diseño, desarrollo y mantenimiento de todo el sistema servidor.

Respecto al coste asociado a las diferentes topologías que hemos analizado, tanto P2P como la topología cliente-servidor host tienen unos costes asociados bajos, estando principalmente asociados al sistema de descubrimiento de jugadores que haga posible la formación de dichas topologías. Al emplear un servidor de reenvío, los costes aumentan debido a la necesidad de mantener el servicio de servidor de reenvío además del servicio de descubrimiento de jugadores. Finalmente, la topología más costosa es la basada en servidor dedicado, debido a la implantación y mantenimiento de un servidor dedicado al alojamiento continuo del juego.

3.3. Llamada a procedimiento remoto

La llamada a procedimiento remoto (*Remote Procedural Call* o RPC) es un sistema empleado en computación distribuida que permite llevar a cabo la ejecución de un fragmento de código en una máquina remota como si de una ejecución local se tratase. De esta forma, la RPC proporciona una capa de abstracción al programador, haciendo transparente para este la administración de los aspectos relativos a la conexión y la gestión de los mensajes. Este mecanismo es ampliamente utilizado en comunicaciones cliente-servidor [20].

El funcionamiento de las RPC se diseñó para que su utilización por parte del programador fuera lo más parecida posible a la llamada a un procedimiento local. Su ciclo de ejecución, el cual puede verse en la Figura 4, empieza con la invocación local de la RPC, proporcionándole los argumentos necesarios para su ejecución como se procedería con una función local. Tras la llamada, dichos argumentos se empaquetan en un mensaje de acuerdo a un formato concreto y el mensaje se envía al servidor. Una vez en el servidor, el mensaje se desempaqueta para extraer los argumentos con los que se llamará a la función. Cuando esta termina la ejecución, se empaquetan los argumentos de salida y el resultado de la función, enviándolos de vuelta al cliente. El cliente desempaquetará el resultado recibido, que será devuelto por la RPC, devolviendo el control al código que la invocó [20][42].

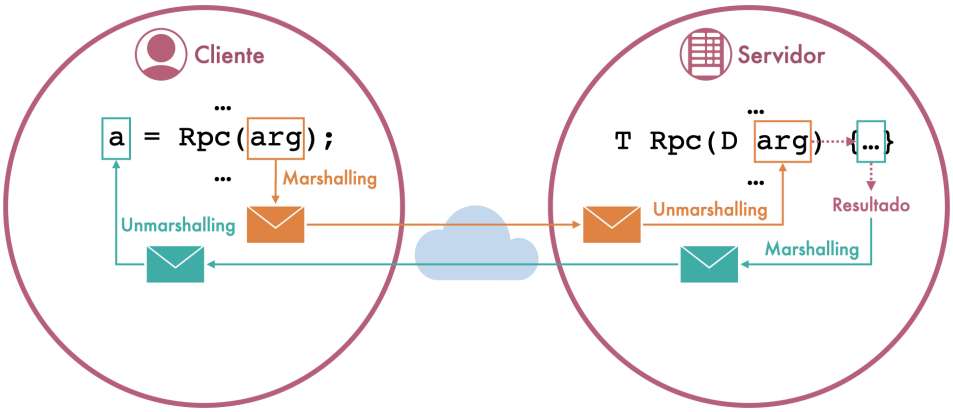

Figura 4: ciclo de ejecución de una RPC.

Las RPC basan el envío de mensajes en el proceso de empaquetado denominado *marshalling*. El marshalling es el proceso por el cual los argumentos a transmitir en la RPC se representan mediante un formato concreto siguiendo unas reglas. Dicho proceso permite evitar que se produzcan problemas de interpretación de los datos transmitidos entre diferentes tipos de hardware, proporcionando un formato estándar intermedio que ambos dispositivos puedan codificar y descodificar para adaptarlos a su propia representación interna [22]. El proceso inverso de desempaquetado o descodificación recibe el nombre de *unmarshalling*.

En el ámbito de los videojuegos multijugador, las RPC ofrecen un mecanismo de sincronización entre clientes y servidores, permitiendo la llamada por parte de clientes de fragmentos de código únicamente disponibles en el servidor, incorporando seguridad y aligerando la carga de cálculo del cliente. Por otra parte, el servidor a su vez también puede llevar a cabo la ejecución de RPC en los clientes, actualizando el estado del juego o de los demás clientes.

4.Metodología

El desarrollo de este proyecto se basa en el empleo de Unity como entorno de desarrollo y motor para nuestro juego [57][53]. Dicha herramienta presenta una serie de ventajas que apoyan su elección:

- Es un entorno de desarrollo multiplataforma que nos permitirá poder llevar a cabo el desarrollo en Windows, Linux o macOS, permitiéndonos también compilar nuestra aplicación para cualquiera de estas plataformas.
- Se trata de una herramienta bien conocida para la cual disponemos de una gran cantidad de documentación y soporte en la red.
- Tiene una gran cantidad de paquetes disponibles con diferentes fines para facilitar el desarrollo, permitiéndonos centrar nuestra atención en la componente multijugador en red de nuestro juego.
- Se trata de una herramienta gratuita en su versión básica, pudiendo acceder también a su versión profesional de forma gratuita al acreditarse como estudiante.

Sin embargo, como hemos expuesto en la sección 1.1, el estado de incertidumbre en que se ha visto envuelta la solución multijugador de esta herramienta en los últimos años supone un reto. Esto se debe a que parte de la documentación relativa a UNet ya no se encuentra disponible, mientras que la documentación de APIs más actuales como MLAPI todavía sigue en desarrollo y es escasa. Esta situación, teniendo en cuenta la gran popularidad de esta herramienta tanto en el aprendizaje de desarrollo de videojuegos como en su uso para producciones comerciales, nos lleva a considerar necesaria la aclaración y exploración del estado y capacidades de dicha herramienta para el desarrollo de videojuegos en red en la actualidad.

Así, el desarrollo del proyecto se ha llevado a cabo haciendo uso de C# sobre una máquina con sistema operativo macOS en su versión 11.4 y la versión 2020.3.10f1 de Unity. Cabe destacar que, por limitaciones de recursos, la mayor parte de los procesos de iteración sobre el proyecto y testeo se han llevado a cabo en una sola máquina haciendo uso de múltiples instancias de la aplicación para probar la capacidad multijugador.

Para tener un seguimiento de las diferentes versiones del proyecto y copias de seguridad de las mismas, hemos empleado la herramienta GitHub [17] como sistema de control de versiones. Esto se debe a las facilidades que ofrece para la creación de repositorios y el etiquetado de versiones, siendo una herramienta muy conocida y cuyo uso durante nuestros estudios ha sido extenso.

La metodología de desarrollo que hemos utilizado se enmarca dentro del ámbito de las metodologías ágiles SCRUM [36][62]. El empleo de este tipo de metodología nos ha permitido una mayor capacidad de adaptación necesaria para la exploración y el estudio de un campo en pleno cambio como es la solución multijugador de Unity. Además, este tipo de metodología, enfocada al desarrollo incremental de funcionalidades que permiten obtener en cada ciclo una versión funcional del proyecto, se adapta a nuestro planteamiento inicial para el desarrollo del mismo. Este planteamiento consiste en cuatro grandes fases o hitos:

- En primer lugar, proceder al desarrollo de una versión local inicial de juego. Esta nos permite desarrollar las mecánicas básicas, que se mantendrán en la versión en red.
- En segundo lugar, proceder al desarrollo de una versión multijugador en línea basada en UNet, permitiéndonos analizar el proceso de adaptación de un proyecto local a uno en línea.
- En tercer lugar, llevar a cabo la ampliación de la funcionalidad multijugador de la versión básica del juego en UNet, aprovechando la conectividad que proporciona un juego en línea. Esta ampliación consistirá en, por una parte, fomentar la competitividad entre los jugadores mediante tablas de resultado y, por otra parte, facilitar la comunicación entre los jugadores gracias a la incorporación de un sistema de nombres de usuario, chat de texto y transmisión de imagen o sonido. De esta manera, los jugadores podrán activar la webcam y/o el micrófono de su dispositivo y transmitir la imagen y/o el audio de los mismos a los demás jugadores.
- En último lugar, proceder a la transición del proyecto desarrollado sobre UNet a la nueva MLAPI. Esta fase nos permitirá contraponer el obsoleto UNet con la solución de más reciente actualidad MLAPI, pudiendo estudiar la compatibilidad entre ambas y el futuro de la solución multijugador de Unity.

Además, dentro de estas cuatro fases, el desarrollo se ha dividido en escenas, abordando progresivamente las diferentes funcionalidades de cada una de las escenas necesarias para componer el proyecto. Hemos podido así, como se detalla más adelante en el capítulo 5, adaptarnos a la dificultades e imprevistos surgidos de la obsolescencia de UNet y el hecho que MLAPI aún se encuentra en desarrollo.

Por tanto, mediante esta metodología, pretendemos obtener como resultado de nuestro desarrollos tres proyectos, correspondientes a las tres fases principales de desarrollo. Así, por un lado, para la versión local queremos obtener un juego básico, con unas mecánicas y un ciclo de interacción sencillos, pero que nos permitan cubrir todos los aspectos esenciales de un juego actual.

Por otro lado, para las versiones en red pretendemos conseguir un juego que implemente el mismo ciclo de interacción y juego que la versión local, pero permitiendo la participación simultánea de múltiples usuarios situados en diferentes máquinas. Un requisito importante del proyecto será permitir un proceso de conexión entre usuarios transparente, evitando que los jugadores deban encargarse de aspectos como determinar quién será el host o introducir la dirección y puerto del mismo. Por tanto, deberá ser el sistema el que automatice este proceso y se encargue de administrar las conexiones de los jugadores.

Como hemos establecido anteriormente, deseamos también enriquecer la experiencia del juego aprovechando la conectividad para fomentar la interacción y la competitividad entre los usuarios. Será importante por tanto, y más en la situación global actual, explorar las capacidades que nos ofrece Unity para permitir la comunicación entre usuarios tanto mediante chat de texto como mediante comunicación audiovisual ya sea a través de UNet o de MLAPI. Finalmente, la Figura 5 muestra tanto el ciclo de interacción local como el ciclo de interacción en línea que buscamos conseguir.

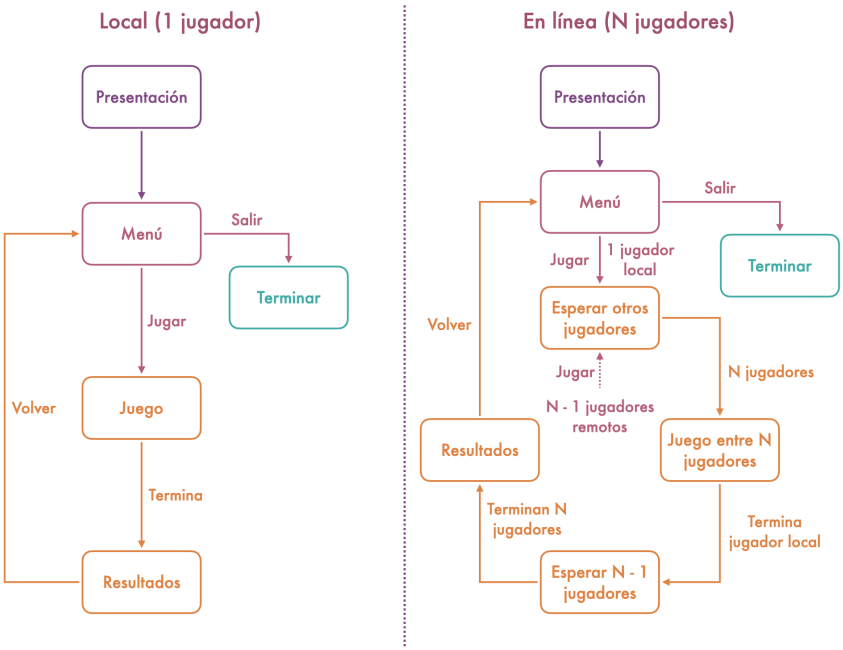

Figura 5: ciclo de juego local (a la izquierda) y ciclo de juego en línea (a la derecha).

5.Desarrollo

El problema que se propone en este trabajo es determinar los pasos necesarios a seguir a la hora de desarrollar un juego multijugador en red con Unity. De esta manera, se pretende concretar en qué se diferencia el desarrollo de un juego local y de un juego en red, proporcionando una referencia para futuros desarrolladores en una etapa de cambio de paradigma en la herramienta.

5.1.Propuesta

La mejor forma para poder determinar las necesidades de un juego multijugador comparadas con uno local es llevar a cabo la conversión a multijugador de un videojuego ya desarrollado en Unity. Para ello, es necesario primero poder partir de un proyecto ya funcional. Por este motivo, el primer paso para el desarrollo será el diseño e implementación de una versión que aborde los aspectos básicos que se esperan en un juego en la actualidad.

Este primer desarrollo nos proporcionará la base a partir de la cual podremos extender el proyecto para permitir la interacción de múltiples usuarios situados en máquinas distintas conectadas a través de la red. Por tanto, el segundo paso consistirá en el diseño e implementación de la funcionalidad multijugador en línea basada en UNet. Para ello comenzaremos dotando al proyecto de la misma funcionalidad básica que la versión local pero permitiendo la conexión de múltiples usuarios y, una vez conseguido, procederemos a ampliar la funcionalidad para explotar las posibilidades que nos ofrece la comunicación en red y fomentar la interacción entre los usuarios.

Para terminar, la última fase del desarrollo consistirá en la adaptación de nuestro proyecto sobre UNet a la nueva MLAPI. Esto nos permitirá el estudio de la más reciente actualidad en el desarrollo de aplicaciones en línea mediante Unity, pudiendo establecer una mejor comparativa entre ambas soluciones.

5.2.Proyecto de partida

El juego que planteamos para llevar a cabo el estudio de la conversión a una versión en línea es un juego de carrera infinita 2D para ordenadores. Por tanto, el objetivo del mismo será recorrer la mayor distancia posible en un terreno infinito hacia la derecha con obstáculos que se generará de forma procedural. La elección de este género nos permitirá reducir las necesidades de diseño de niveles y creación de *assets* gracias a su estilo bidimensional y su generación procedural, permitiendo un mayor enfoque en nuestra tarea principal que es el diseño e implementación de la lógica en red del mismo.

5.2.1.Concepto

En el diseño del concepto de este proyecto hemos tratado de dar prioridad a la simplicidad de las mecánicas e interacciones para poder centrar nuestra atención en el desarrollo y transición a la versión en red del juego. Así, las mecánicas principales del juego que planteamos son:

- Desplazarse a izquierda y derecha mediante las flechas izquierda y derecha del teclado.
- Saltar mediante la barra espaciadora.
- Interactuar con elementos del entorno mediante la tecla Z.

Se plantea también un interfaz basado en una sucesión de pantallas, representado en la Figura 6, por el cual el usuario comienza en un menú principal mediante el cual se puede cerrar el juego o comenzar a jugar. Si la opción de jugar se selecciona, pasamos a la pantalla de la sala de introducción, donde el jugador toma control de un pequeño cuadrado, el cual será el avatar del jugador. En dicha escena cerrada, habrá también dos elementos, a modo de puertas, con los que el jugador podrá interactuar: el primero permitirá al jugador regresar al menú principal, mientras que el segundo le permitirá iniciar una nueva carrera. La presencia de esta sala no es estrictamente necesaria en la versión local de nuestro juego y se podría omitir, pero constituirá un elemento fundamental de la versión en línea, actuando como punto de encuentro y socialización para nuestros jugadores. Por este motivo, añadimos la sala de introducción como precursora al futuro *lobby* en la versión local.

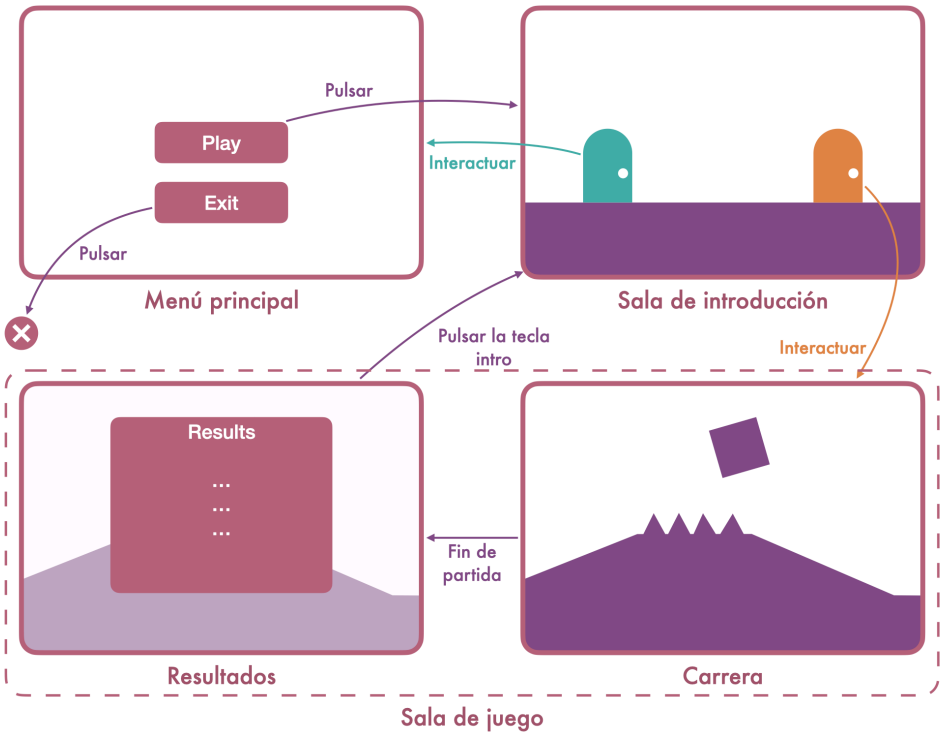

Figura 6: diagrama de navegación entre pantallas.

La carrera presentará al jugador sobre un terreno ascendente y descendente en el que, sobre el suelo, podemos encontrar púas que dañan al jugador al tocarlas. Sobre la carrera se presenta una sencilla interfaz en la que el jugador puede ver su barra de salud, la distancia recorrida y el tiempo transcurrido. La carrera se podrá pausar al pulsar la tecla tabulador y, cuando la vida del jugador llegue a cero, la partida se detendrá y se mostrará sobre la carrera una interfaz indicando al jugador la distancia recorrida. En este punto, al presionar la tecla intro el jugador volverá a la sala de introducción. En la Tabla 2 podemos ver el esquema de controles resultante.

Controles	
Tipo de interacción	Método de interacción
Movimiento del jugador.	Flechas izquierda / derecha. Teclas A / D.
Interacción con las puertas.	Tecla Z.
Salto.	Barra espaciadora.
Pausar / reanudar durante la carrera.	Tabulador.
Volver a *lobby* tras la carrera.	Intro.
Interacción con el menú principal.	Ratón.

Tabla 2: esquema preliminar de controles del juego.

La pantallas del menú principal y la sala de introducción tendrán sus propias escenas, mientras que la carrera y la pantalla de resultados compartirán la misma escena. Por tanto, a continuación analizaremos los componentes que van a formar cada una de estas escenas y discutiremos la implementación de los mismos.

5.2.2. Menú principal

El menú principal requiere dos botones que permitan comenzar la partida y cerrar el juego. Por tanto esta escena solo necesita un lienzo, al cual asignaremos dos botones como hijos. Para implementar la interacción con los botones necesitaremos definir un objeto UIManager al que le asociamos un *script* *MainMenuManager.cs*, el cual proporciona una función `PlayPressed()` que cargará la sala de introducción y una función `ExitPressed()` que cerrará el juego. Con estas funciones, bastará con añadirlas en el editor a los eventos de los respectivos botones para dotar a la interfaz de la funcionalidad necesaria. En la Figura 7 podemos ver la escena resultante.

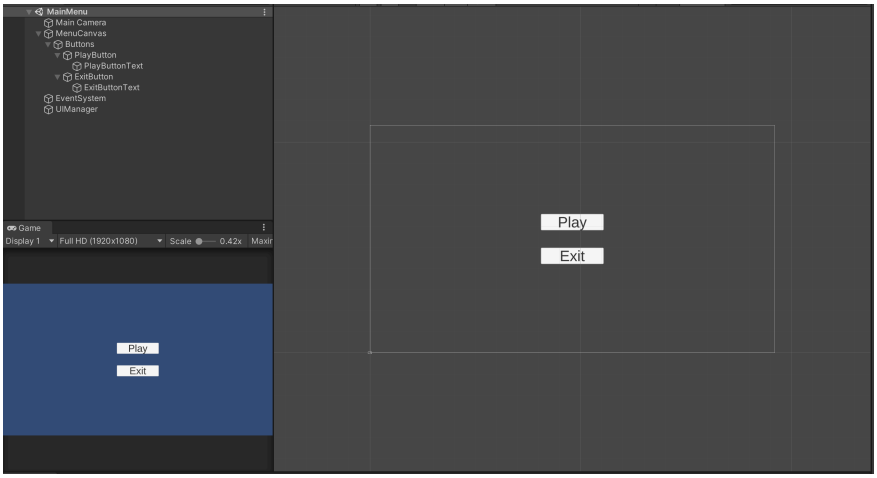

Figura 7: captura de la escena del menú principal en el editor de Unity.

5.2.3.Sala de introducción

La sala de introducción es la primera escena en la que se toma el control del avatar del jugador. Por este motivo, debe contener:

- El avatar del jugador, que gracias al *script Player.cs* podrá ser controlado haciendo uso de las flechas del teclado y la barra espaciadora. Será necesario también que el objeto jugador tenga asignado los componentes BoxCollider2D y Rigidbody2D, ya que estos permitirán la simulación física del mismo, siendo además necesarios para el correcto funcionamiento del *script*.

- Tres rectángulos a modo de suelo y paredes, cada uno con los correspondientes BoxCollider2D para asegurar que no serán atravesados por el jugador.

- Los objetos encargados de iniciar la carrera (GameStarter) y cerrar la sala de introducción (LobbyExit). Para hacer posible la interacción con objetos por parte del jugador, creamos la interfaz `InteractableObject`, la cual definirá los métodos necesarios para la interacción. La clase `SceneLoader` implementará dicha interfaz para cargar la escena indicada al ejecutar el método `Interact()` de la interfaz y devolver el código de la tecla de interacción al ejecutar el método `GetKey()` de la interfaz. De esta forma, gracias al uso de esta clase, la etiqueta "Interactable" y un BoxCollider2D que actúa como disparador, el jugador podrá interactuar con GameStarter y LobbyExit. Extendemos por tanto la clase `Player` de tal forma que en la llamada al método `OnTriggerEnter2D(…)` se almacene la referencia al objeto atravesado si tiene la etiqueta "Interactable". Así, si se tiene una referencia a un objeto interactivo, cuando se detecte que la tecla indicada por `GetKey()` ha sido presionada, se ejecutará el método `Interact()` del objeto referenciado. De igual forma, el método `OnTriggerExit2D(…)` se encargará de limpiar la referencia al objeto interactivo. Finalmente, GameStarter se ha representado visualmente mediante un cuadro rojo, mientras que LobbyExit se ha representado mediante un cuadrado azul, como puede verse en la Figura 8.

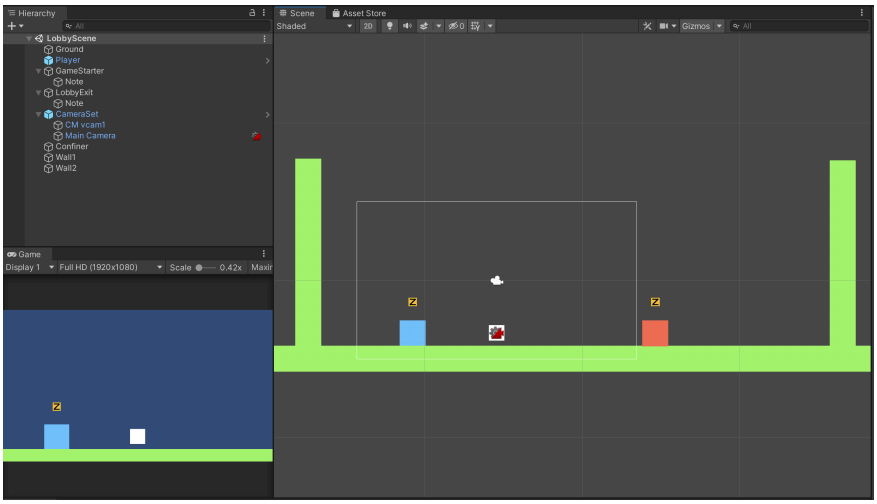

Figura 8: captura de la escena de la sala de introducción en el editor de Unity.

- Finalmente, será necesario disponer de una cámara que sea capaz de seguir al jugador, por lo que haremos uso de una cámara virtual Cinemachine, la cual configuraremos para que lleve a cabo su seguimiento y limite su propio movimiento

de tal forma que la cámara no pueda salirse de los límites del nivel. Para disponer de este tipo de cámara, será preciso instalar Cinemachine a través del administrador de paquetes de Unity. La Figura 9 resume los objetos que forman la sala de introducción.

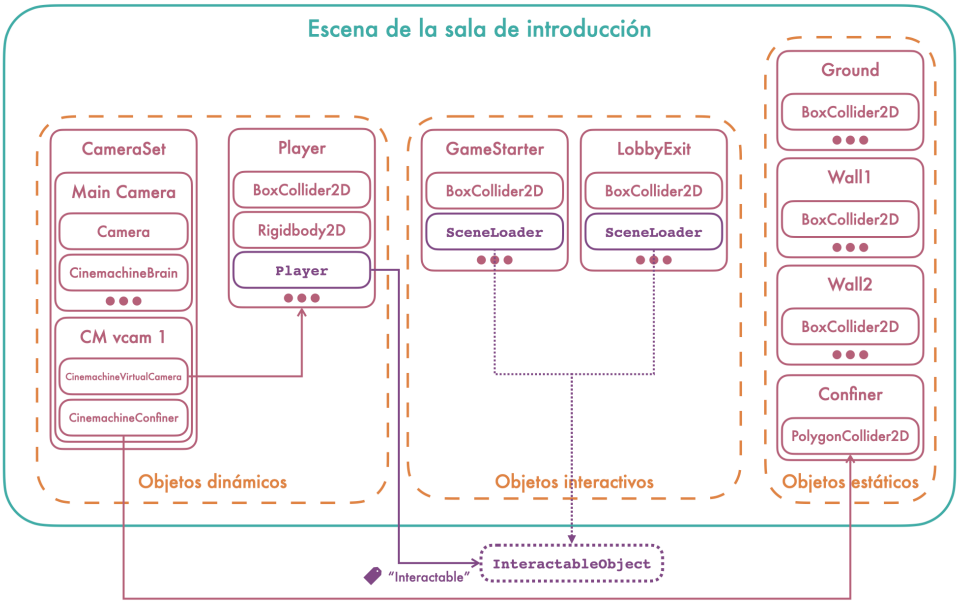

Figura 9: esquema resumen de los objetos de la sala de introducción.

5.2.4.Carrera y pantalla de resultados

En la escena correspondiente a la carrera tendremos, por un lado, los elementos propios de la lógica del juego y, por otro, los elementos que conforman la interfaz. Los elementos que conforman la lógica de juego son:

- El objeto CellGenerator, el cual será el responsable de la generación procedural del suelo de la carrera gracias al *script CourseGanerator.cs*. Para poder llevar a cabo esta generación, se concibe el suelo como una matriz de celdas de terreno que deben cumplir una serie de reglas de vecindad. Así, este *script* se encarga de generar el suelo de forma aleatoria a medida que la cámara principal avanza hacia la derecha.

  Cuando la distancia entre la cámara y el generador es menor que un margen dado, el generador producirá nuevas columnas de celdas hasta que la distancia sea igual o superior al margen, ya que la posición del generador siempre coincidirá con la posición de la última celda generada. La generación de una columna dependerá de la columna anterior, asegurando que las reglas de vecindad entre las columnas se cumplen y dada una columna, se determinará la altura máxima de la columna siguiente. De esta manera, será la generación aleatoria de celdas la que determinará, dada la altura máxima, si la columna que está siendo actualmente generada será una unidad más pequeña o una unidad más grande que la anterior, consiguiendo generar un terreno ondulante en su vertical.

  Para definir las reglas de vecindad, diseñamos cinco tipos de celdas diferentes que guardamos como *prefabs*: FullCell, RampUpCell, RampDownCell, SpikesCell y EmptyCell. Estas representarán respectivamente una celda completamente llena de terreno, una celda con una rampa ascendente, una celda con una rampa descendente,

una celda con púas y finalmente una celda vacía sin terreno. Así, cada una de ellas definirá las conexiones que requiere, las cuales pueden verse en la Figura 10 representadas mediante una línea punteada azul. Por ejemplo, RampUpCell necesitará una conexión abajo y una conexión a la derecha, por lo que únicamente podrá ir encima de una celda con una conexión superior y a su derecha será necesario que haya una celda con una conexión a la izquierda. El modelado de estas celdas y sus conexiones se lleva a cabo gracias al *script Cell.cs* que todas ellas incluyen.

De esta manera, en *CourseGenerator.cs* se generarán las columnas de arriba a abajo, rellenando cada columna con celdas vacías hasta llegar a la altura máxima deseada. Una vez se alcance la altura máxima posible para la columna, la generación permitirá la inserción de celdas diferentes a la celda vacía, analizando siempre la celda de la izquierda de la posición actual para asegurar la correcta conexión entre columnas. Si la celda propuesta para una posición no es compatible, se generará una nueva hasta encontrar una celda apropiada para dicha posición. Finalmente, una vez generada la primera celda con conexión abajo, el resto de celdas se rellenarán con celdas llenas.

Cabe también la posibilidad de que al llegar a la altura máxima posible se genere una celda de púas, en cuyo caso se comprobará que el número de púas situadas de forma consecutiva no supere un valor máximo dado. Las púas utilizan la etiqueta “Damage” y todas las celdas, a excepción de las celdas vacías, tienen los colisionadores necesarios para que el jugador no los pueda atravesar.

El objeto CellGenerator permite configurar los parámetros de su *CourseGenerator.cs*, pudiendo configurar el tamaño de las celdas, el número máximo de púas consecutivas, el número de filas y el conjunto de celdas empleadas para la construcción del suelo. De esta manera, la Figura 10 contiene el diagrama de flujo correspondiente al algoritmo de generación de terreno y en la Figura 11 podemos ver un ejemplo de terreno generado.

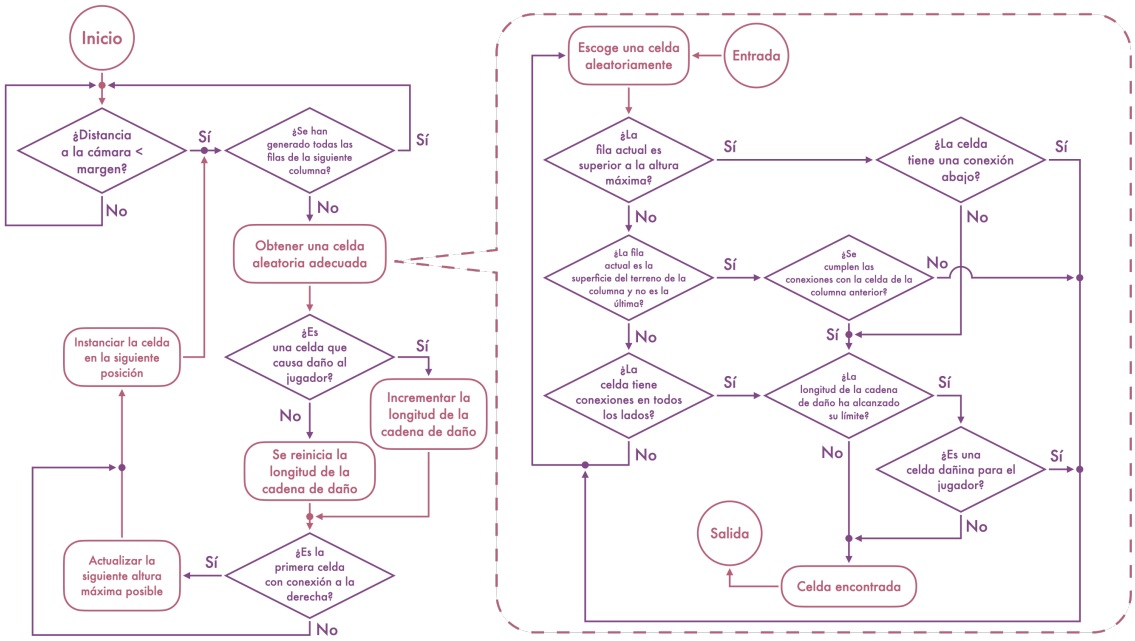

Figura 10: diagrama de flujo del algoritmo de generación de terreno.

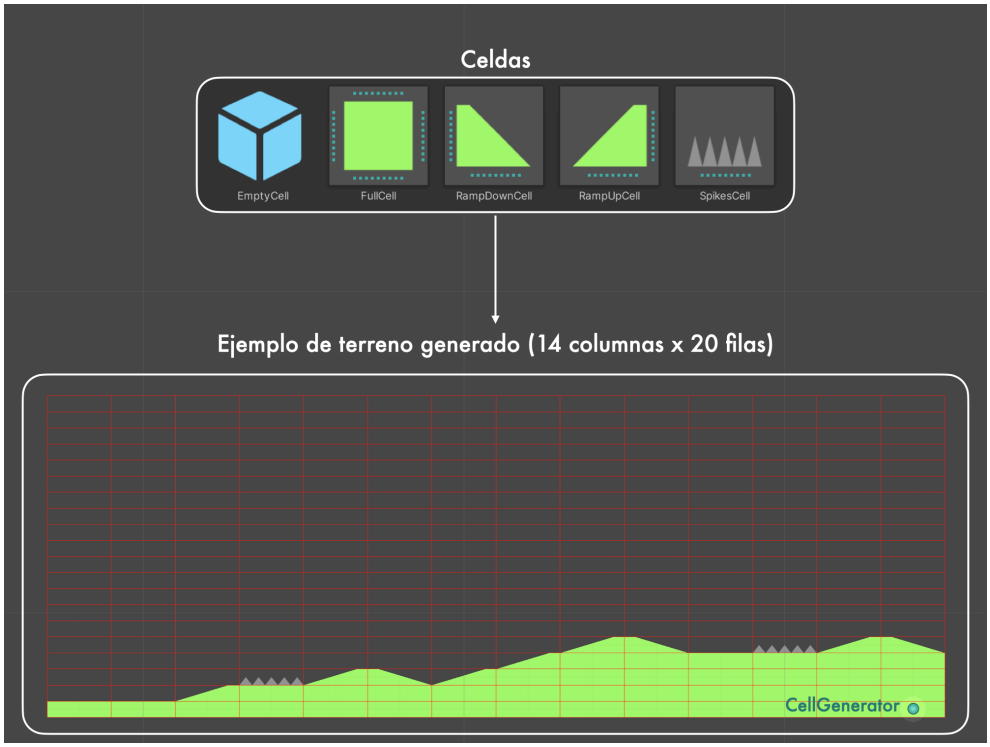

Figura 11: ejemplo de generación de terreno y las celdas empleadas para la misma.

- El objeto GameController que se encargará de controlar el estado del juego gracias al *script GameController.cs*. Dicho *script* monitorizará el estado del jugador y su distancia recorrida.

  Para poder llevar a cabo este control, hemos hecho uso de la clase `Messenger` diseñada por Magnus Wolffelt y Julie Iaccarino con citas de Rod Hyde [9]. Dicha clase nos permite el intercambio de mensajes internos entre los componentes del juego, siendo posible suscribirse a los mensajes asociados a un evento concreto.

  De esta manera, el *script GameController.cs* registra los eventos asociados al inicio de la carrera (`GameEvent.PLAYER_STARTS`), a la muerte de jugador (`GameEvent.PLAYER_DIES`) y al movimiento del jugador (`GameEvent.PLAYER_MOVED`). Gracias a la recepción de estos eventos es posible la monitorización de la distancia recorrida y el estado del juego. Cuando el jugador muera, `GameController` guardará la puntuación lograda y determinará si esta constituye un nuevo récord, en cuyo caso emitirá un mensaje asociado al evento `GameEvent.NEW_HIGHSCORE_REACHED`. `GameController` también se encargará de pausar/reanudar el juego al pulsar la tecla tabulador, emitiendo los mensajes asociados a `GameEvent.PAUSE` y `GameEvent.RESUME` respectivamente. Finalmente, cuando la distancia recorrida aumente enviará un mensaje con la nueva distancia asociado al evento `GameEvent.DISTANCE_INCREASED`. El empleo de la clase `Messenger` nos permite así incrementar el grado de independencia entre los diferentes componentes.

- El avatar del jugador, el cual podemos reutilizar de la sala de introducción gracias a su almacenamiento en el proyecto como *prefab*. Será necesario extender el *script Player.cs* para almacenar su salud máxima y actual como campos, así como también

implementar la reducción de la salud actual al tocar cualquier objeto etiquetado como "Damage" gracias al método `OnCollisionEnter2D(…)`.

De igual forma, al llegar su salud a cero el jugador emitirá el mensaje asociado al evento `GameEvent.PLAYER_DIED` y de forma periódica en cada `FixedUpdate()` emitirá un mensaje con su posición horizontal asociado al evento `GameEvent.PLAYER_MOVED`. También emitirá el mensaje asociado a `GameEvent.PLAYER_HEALTH_CHANGED` cuando reciba daño y su salud resultante sea mayor que cero, proporcionando el porcentaje de salud resultante en el mensaje.

- La cámara utilizada es la misma que en la sala de introducción gracias a su almacenamiento como *prefab* que permite su reutilización.

- El objeto `Floor` contiene un rectángulo a modo de pared trasera, uno a modo de suelo y el confinador para la cámara. Este conjunto se desplazará siempre hacia la derecha con el movimiento del jugador gracias al *script ForwardHorizontalDisplacement.cs* y su finalidad será limitar el movimiento del jugador hacia la izquierda, clarificando el sentido de avance de la carrera y permitiendo la destrucción de las celdas de terreno ya superadas.

Por otro lado, los elementos que forman la interfaz son:

- Un lienzo CanvasUI que contiene como hijos la barra de salud, el cronómetro y la distancia recorrida. La actualización de estos elementos se lleva a cabo gracias al diseño de la clase `CourseUI` para CanvasUI, la cual registra los mensajes asociados al paso del tiempo (`GameEvent.TIME_PASSED`), el incremento de la distancia recorrida (`GameEvent.DISTANCE_INCREASED`) y el cambio en la salud del jugador (`GameEvent.PLAYER_HEALTH_CHANGED`). El tamaño horizontal del contenido de la barra de salud vendrá dado por el porcentaje de salud actual del jugador.

- Un objeto PausedScreen como hijo de CanvasUI para indicar al jugador visualmente la activación de la pausa. La activación y desactivación de PausedScreen es controlada por CanvasUI al recibir los mensajes asociados a `GameEvent.PAUSE` y `GameEvent.RESUME` respectivamente enviados por `GameControler`.

- Un objeto FinishiedScreen como hijo de CanvasUI que constituirá la interfaz de resultado de la carrera. Esta interfaz mostrará la distancia recorrida gracias a la clase `ScoreScreen`, la cual escuchará los mensajes asociados al evento `GameEvent.NEW_HIGHSCORE_REACHED` y cargará la distancia recorrida durante la carrera. La activación de FinishedScreen es controlada por CanvasUI al recibir el mensaje asociado a `GameEvent.PLAYER_DIED` y, una vez activo, FinishedScreen se procederá a cargar la sala de introducción al pulsar la tecla intro gracias a `ScoreScreen`.

Con todo esto ya tenemos nuestro proyecto de partida para poder abordar el proceso de conversión a un juego multijugador en red. Como se puede ver en las Figuras 7, 8 y 12, los gráficos del juego para este proyecto de partida se han mantenido simples para centrar el énfasis y enfoque del proyecto en la implementación multijugador, considerando el desarrollo de gráficos más atractivos una tarea secundaria. Por otra parte, la Figura 13 presenta el diagrama de los objetos que componen la escena de la carrera.

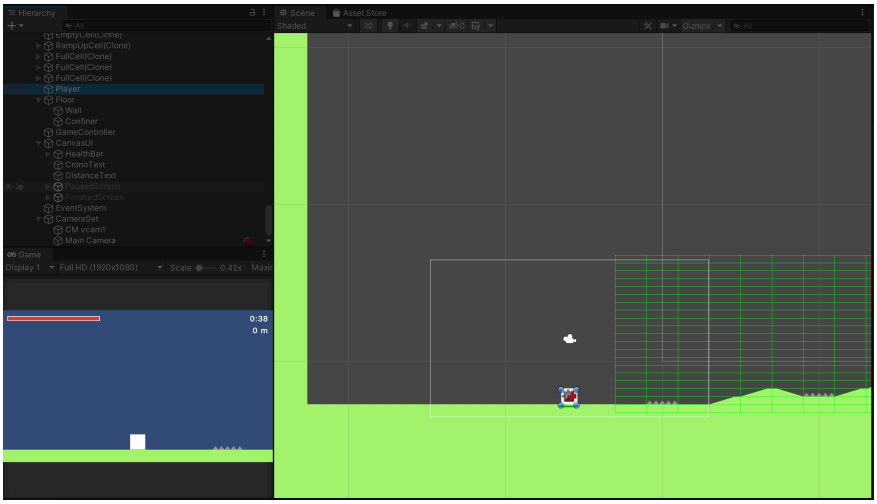

Figura 12: captura de la escena de la carrera en ejecución dentro del editor de Unity.

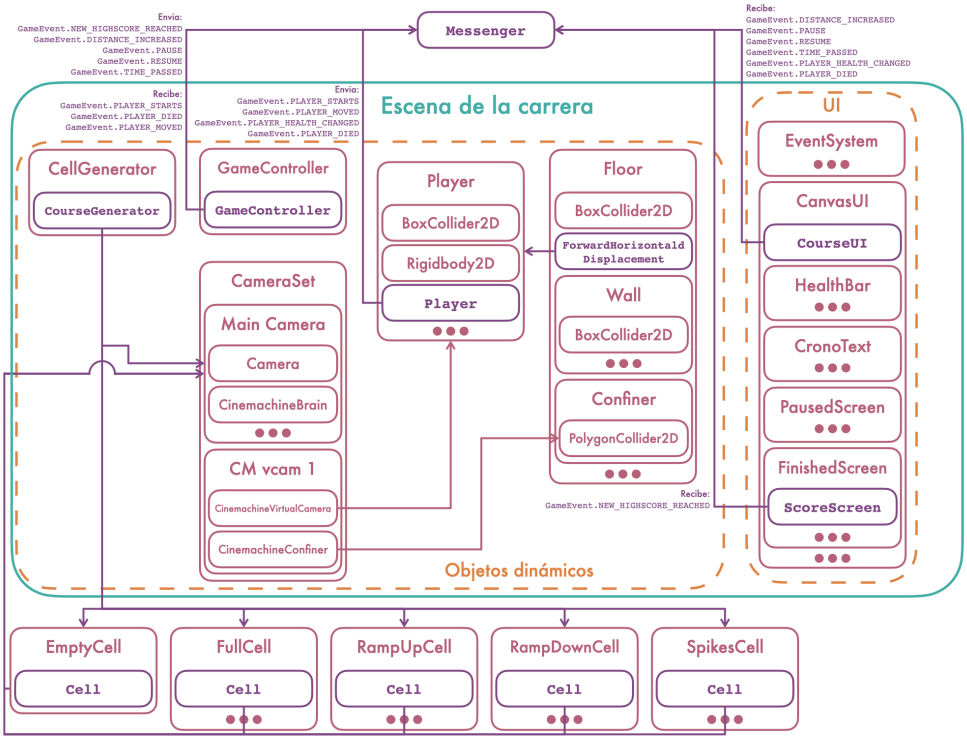

Figura 13: esquema resumen de los objetos de la escena de carrera.

5.3.Adaptación con UNet: carrera infinita multijugador

A continuación procederemos a ver cómo la HLAPI de UNet permite el desarrollo de juegos en red. Para ello, nos centraremos en el ejemplo local ya desarrollado, definiremos la funcionalidad que deseamos conseguir para su versión multijugador y finalmente procederemos al diseño e implementación de la misma.

5.3.1.Concepto

La funcionalidad que se desea conseguir al añadir la dimensión multijugador al juego es permitir al usuario reunirse con otros jugadores para poder llevar a cabo

competiciones, pero también como punto de encuentro para socializar. Por estos motivos, se concibe la sala de introducción como un *lobby* en el que los jugadores se reunirán pudiendo interactuar mediante un chat de texto o la transmisión en vivo de audio y vídeo. En cualquier momento, un jugador podrá dar inicio a una carrera, en cuyo momento se dará un tiempo para que otros jugadores puedan unirse. Es importante permitir que los jugadores que no quieran proceder a jugar una carrera en ese momento puedan quedarse juntos en el *lobby*, mientras que los que quieran correr puedan empezar juntos la carrera.

Una vez dentro de la carrera, los jugadores podrán ver el progreso de los demás, pudiendo ver su salud y distancia recorrida. Cuando un jugador muera, podrá observar a los otros jugadores en su partida y cuando todos hayan terminado, se mostrará a cada uno de ellos una lista con los resultados de todos los jugadores. Tras finalizar la carrera, los usuarios podrán volver al *lobby* donde se podrán encontrar con otros jugadores.

5.3.2. Diseño

Primero, determinaremos el nivel al que pertenece nuestro juego multijugador según los diferentes tipos detallados en la sección 3.1. Nuestra idea de juego implica la realización de carreras en tiempo real a modo de partidas, pudiendo además permitir la comunicación entre los jugadores a través de diferentes medios. Esto nos sitúa dentro de la categoría de juegos en red en tiempo real L3. A continuación, y teniendo en cuenta las topologías vistas en el apartado 3.2, planteamos una topología cliente-servidor host como base para el diseño. De esta manera, uno de los jugadores tomará el papel de servidor y cliente (host), mientras que el resto de jugadores actuarán únicamente como clientes. El motivo de esta elección reside, por un lado, en el diseño de UNet para permitir topologías de tipo host o servidor y, por otro, en el hecho de que el diseño de este tipo de topología es fácilmente extensible a una topología con servidor de reenvío o servidor dedicado. Así, esto nos permitirá abordar todos los elementos principales de un juego multijugador y un nivel de sincronización, entre los mismos, exigente por las restricciones de tiempo real.

La conexión con otros jugadores debe producirse en el *lobby*, siendo por tanto el menú principal una escena que no requiere conexión con ningún host. El encuentro entre los jugadores en el *lobby* debe producirse de forma natural y transparente para el jugador, evitando el tener que introducir manualmente parámetros como la dirección IP o el puerto del jugador que está actuando como host. Por este motivo, será necesario la presencia de un sistema de descubrimiento que permita encontrar hosts activos, como se muestra en la Figura 14. Debido a las restricciones causadas por la falta de un servidor que nos permita llevar a cabo el emparejamiento de clientes y hosts, asumiremos (sin pérdida de generalidad) que todos los jugadores se encontrarán conectados a la misma red local. De esta manera, el sistema de descubrimiento iniciará su funcionamiento en modo cliente, tratando de encontrar un host al que conectarse, en cuyo caso se conectará al primer host que se encuentre. Si el sistema no encuentra ningún host, entonces se convertirá él mismo en uno, notificando su presencia a otros clientes que busquen un host.

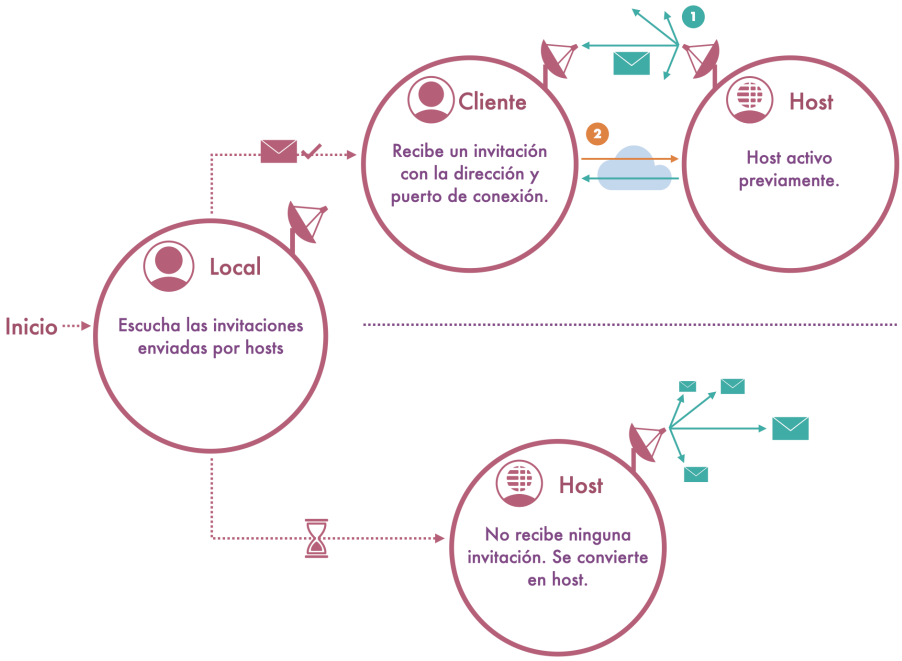

Figura 14: esquema de funcionamiento del sistema de descubrimiento.

Una vez los clientes se encuentren conectados a un host, cuando un jugador inicie una carrera se formarán dos grupos: el grupo de juego que contendrá los jugadores que procederán a correr, y el grupo de *lobby* que contendrá los jugadores que no van a correr. Por tanto, hay que considerar dos posibles escenarios cuando se inicie una partida, los cuales se representan en las Figuras 15 y 16:

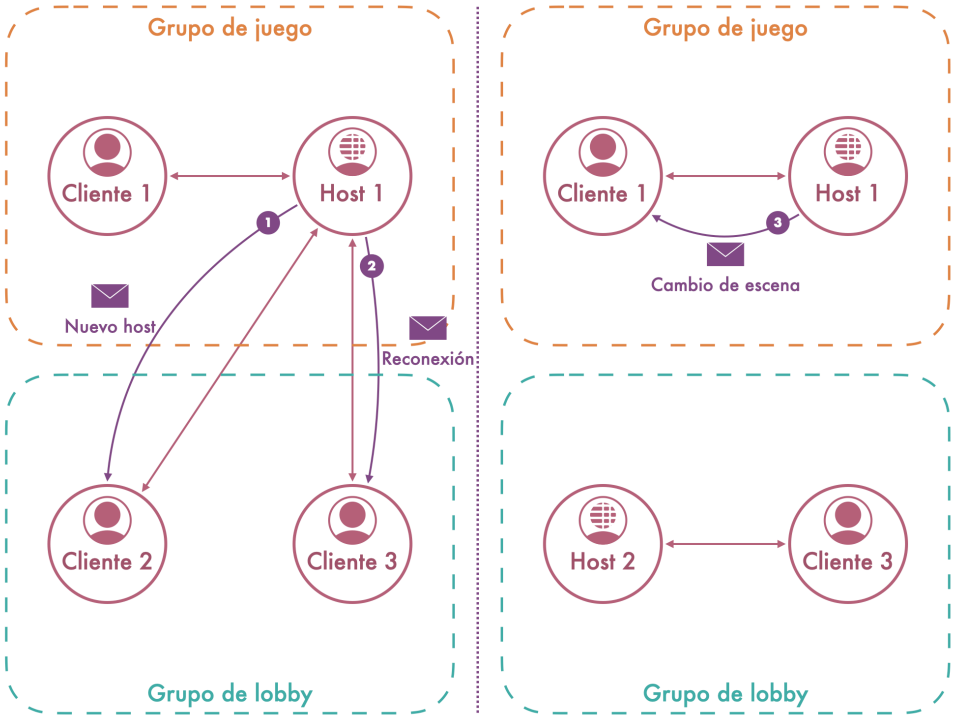

Figura 15: esquema de separación de jugadores cuando el host está en el grupo de juego.

- El host actual (A) se encuentra en el grupo de juego. En este caso, para conseguir que el resto de jugadores pueda quedarse en el *lobby*, será necesario nombrar un nuevo host (B) de entre el grupo de *lobby*. De esta manera, se escogerá el primer miembro del grupo de *lobby*, el cual se desconectará del host actual (A) y se iniciará como host del *lobby* (B). El resto de miembros del grupo de *lobby* se desconectarán del host actual (A) para conectarse al nuevo host del *lobby* (B). Finalmente el host actual (A) llevará el cambio de escena para todos los miembros del grupo de juego, pasando a ser el host de la carrera.

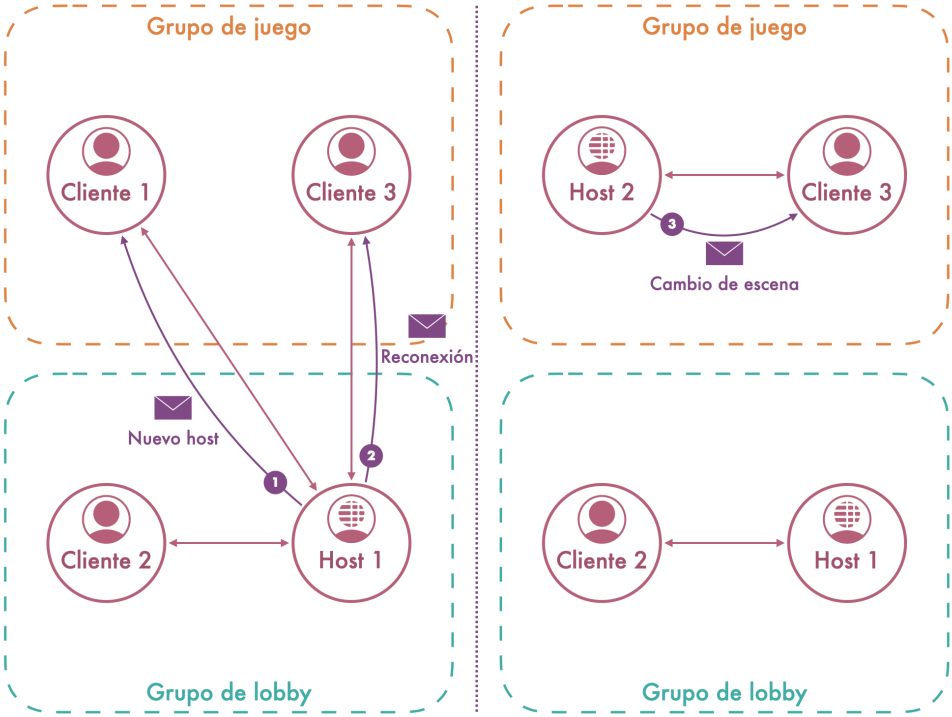

Figura 16: esquema de separación de jugadores cuando el host no está en el grupo de juego.

- El host actual (A) se encuentra en el grupo de *lobby*. En este caso, para conseguir que los jugadores del grupo de juego puedan comenzar la carrera, será necesario nombrar un nuevo host (B) para la misma. De esta manera, se escogerá el primer miembro del grupo de juego, se desconectará del host actual (A), y se iniciará como host de la carrera. El resto de miembros del grupo de juego se desconectarán del host actual (A) y se conectarán al nuevo host (B). Cuando el nuevo host (B) reciba tantas conexiones como miembros había en el grupo de juego, procederá a la carga de la escena de carrera para todos los jugadores conectados.

Una vez comenzada la carrera, el host será el encargado de la generación procedural del terreno, el cual se sincronizará con cada uno de los clientes. En la versión local la generación del terreno depende de la posición de la cámara del jugador. En cambio, en la versión en red, el sistema de generación no podrá depender de la posición de la cámara del host ya que este podría ir en última posición y causar que no se genere el terreno para los jugadores de las primeras posiciones. Por tanto, será necesario conocer cuál es el jugador que va en primera posición y conocer el valor de la misma, ajustando la generación del terreno a esta. De igual manera, la destrucción del terreno generado ya recorrido dependerá de la posición del jugador situado en la última posición.

Finalmente, cuando todos los jugadores agoten su salud y la carrera termine, cada uno de los clientes conectados al host de la carrera se desconectará. Una vez todos los clientes se hayan desconectado, el host también se cerrará. Cuando el jugador presione la tecla intro, se cargará el *lobby* y el proceso de descubrimiento volverá a iniciarse. Esto permite la reagrupación de los jugadores que estaban en la carrera con los jugadores que se quedaron en el *lobby*. Este proceso puede verse en la Figura 17.

En cuanto a la desconexión de los jugadores, habrá que determinar en el momento de desconexión si se está actuando como cliente o como host. Si estamos actuando como cliente, bastará con informar al host antes de proceder a la desconexión. No obstante, si estamos actuando como host, será necesario escoger un nuevo host de entre los jugadores actuales y proceder a la migración de todos los clientes al nuevo host, de forma similar al esquema empleado para comenzar una carrera.

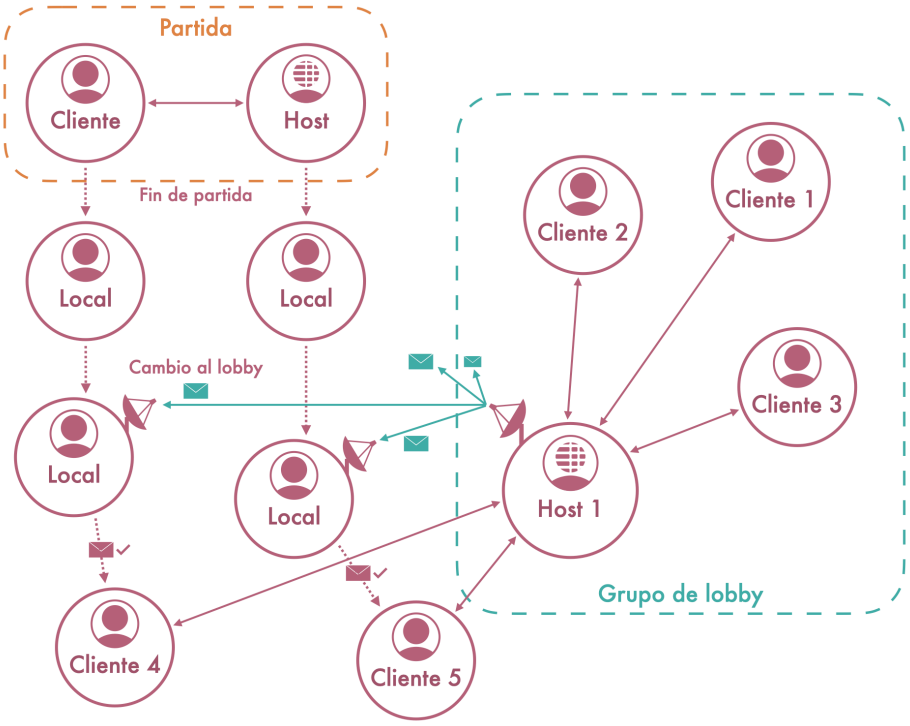

Figura 17: esquema de reconexión con el *lobby* al finalizar la partida.

5.3.3. Propuesta de implementación

Un paso muy importante para este proceso es determinar qué objetos será necesario sincronizar a través de la red y qué objetos podrán existir como copia local en cada dispositivo. A la hora de tomar esta decisión hay que tener en cuenta aspectos como en qué dispositivos debe estar presente el objeto, si es estático o dinámico y sus necesidades de compartir información entre diferentes jugadores.

5.3.3.1. Instalación de Multiplayer HLAPI

El primer paso necesario para la implementación es la instalación del paquete HLAPI de UNet en nuestro proyecto. Para ello, abriremos el administrador de paquetes y buscaremos de entre los paquetes disponibles en *Unity Registry* el paquete *Multiplayer HLAPI*, mostrado en la Figura 18. Una vez nos aparezca, bastará con hacer clic en el botón *Install* para añadirlo a nuestro proyecto.

Con la HLAPI instalada, ya podemos proceder a convertir nuestro proyecto en un juego multijugador en red. Para ello, empezaremos convirtiendo la sala de introducción en el *lobby* de nuestro juego, detallando los pasos necesarios.

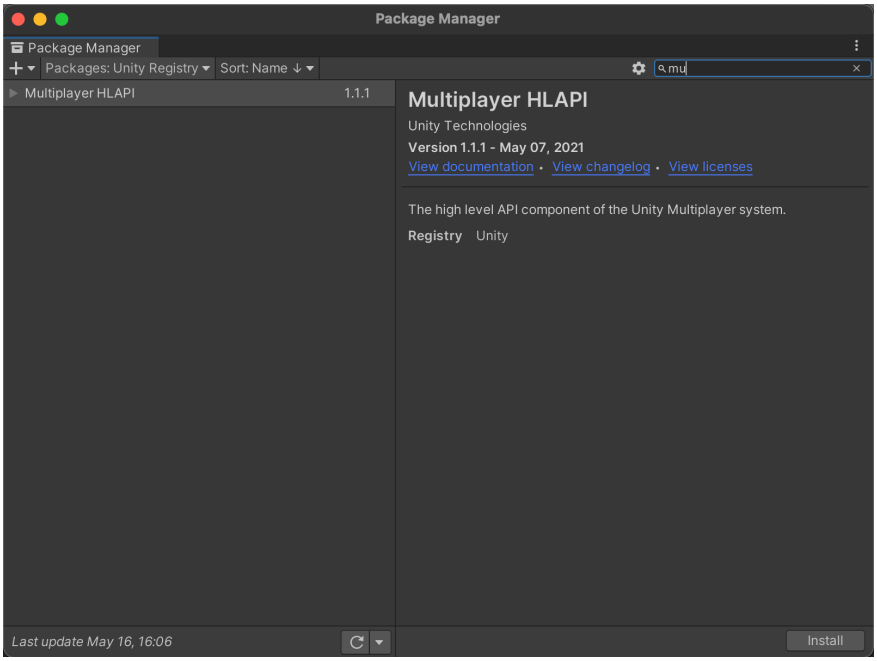

Figura 18: captura de Multiplayer HLAPI en el administrador de paquetes.

5.3.3.2.Lobby

Al analizar los objetos presentes en la sala de espera, encontramos:

- El avatar del jugador, cuyas acciones deberán reflejarse en los dispositivos de los demás jugadores. Por tanto, será un objeto a sincronizar a través de la red.
- La cámara, la cámara virtual Cinemachine y el confinador para la misma deberán ser objetos locales, ya que cada usuario tendrá su propia vista de juego según la posición de su avatar.
- Las paredes y el suelo deben ser las mismas para todos los jugadores, pero ya se encuentran presentes en la escena desde el inicio y nunca ven alterado su estado, por lo que serán objetos locales a cada máquina al carecer de necesidades de sincronización.
- El objeto LobbyExit, encargado de cerrar el *lobby* y volver al menú principal. Este objeto deberá detener la conexión del jugador. En primera instancia, este objeto podría ser local a cada máquina ya que únicamente depende de las acciones del jugador local. No obstante, para poder llevar a cabo la desconexión es necesario conocer si nos encontramos en un cliente o si somos el host, por lo que finalmente consideramos más adecuado que sea un objeto en red para tener acceso al atributo `isServer` de la clase `NetworkBehaviour`.
- El objeto GameStarter, encargado de registrar la participación de los jugadores en la carrera e iniciar la misma tras una cuenta atrás. Para poder registrar todos los

jugadores que van a correr en el servidor, será necesario que GameStarter pueda operar sobre el host, y por tanto deberá ser un objeto en red.

Con esto ya podemos comenzar con la adaptación de la sala de introducción en un *lobby*. El primer paso para esto será crear un gestor de red. Para ello, crearemos en la escena un objeto vacío al que llamaremos NetworkManager, y añadiremos a este el componente NetworkManager de HLAPI. Este componente es el que permitirá la conexión con otros jugadores mediante el inicio como cliente, host (cliente y servidor) o servidor. Además, almacenará aspectos necesarios como la IP y el puerto de conexión.

Para facilitar el desarrollo, HLAPI ofrece el componente NetworkManagerHUD, que al ser añadido a NetworkManager nos permitirá el control del mismo, pudiendo usarlo para iniciar un cliente, un host o un servidor. Esta pequeña interfaz, presente en la Figura 19, solo tiene como objetivo dar soporte al desarrollo y al testeo, y se supone que será reemplazada por el desarrollador en el lanzamiento de su aplicación. En nuestro caso, emplearemos este componente durante las primeras fases del desarrollo hasta que el sistema de descubrimiento de jugadores esté listo.

Figura 19: interfaz proporcionada por NetworkManagerHUD.

El segundo paso será adaptar los diferentes objetos de la escena para que puedan ser sincronizados a través de la red. Hay que destacar que para que un objeto pueda ser compartido y sincronizado entre los jugadores, es necesario que este objeto incluya un componente NetworkIdentity, el cual asignará un identificador único (`netId`) a cada objeto en red, de forma que todas sus instancias tendrán el mismo `netId`. Así también, los *scripts* asociados a objetos en red deberán derivar de la clase `NetworkBehaviour` en lugar de `MonoBehaviour` como los objetos locales.

Primero, comenzamos adaptando el avatar del jugador. Para ello, será preciso que este incluya el componente NetworkIdentity, el cual deberá tener activada la opción *Local Player Authority*. Esta opción es fundamental para dar permiso al jugador para alterar las instancias de sí mismo en otros clientes. Todo objeto que deba poder modificar sus instancias en los diferentes clientes deberá tener esta opción activada. Como el movimiento del jugador se deberá sincronizar entre los diferentes clientes, será necesario también añadir al avatar el componente NetworkTransform, definiendo el modo de sincronización a *Sync Rigidbody 2D* y con la opción *Sync Angular Velocity* activada. Este componente permitirá la sincronización del *rigidbody* de todas las

instancias del avatar a través de la red. Finalmente, llevamos a cabo los cambios pertinentes en el código *Player.cs*:

- La clase `Player` heredará de `NetworkBehaviour` en lugar de `MonoBehaviour`.
- El código relativo a la inicialización, la actualización del estado en función de la entrada del usuario y la reacción al contacto con otros objetos debe llevarse a cabo únicamente si el código se está ejecutando en la máquina del jugador que lo controla. Una de las principales dificultades a la hora de implementar objetos que se distribuyen a través de la red en Unity es que cada una de las instancias del mismo objeto ejecutarán el mismo *script*. Por tanto, cuando un jugador se conecta a un host, tanto la instancia local en la máquina del jugador como la instancia situada en el host ejecutarán el mismo *script Player.cs*. Esto implica que será necesario para todos los objetos distribuidos considerar cuál deberá ser el comportamiento de las diferentes instancias dependiendo de dónde se encuentren y codificar los diferentes casos en el *script* de la entidad distribuida. En el caso concreto del avatar del jugador, las acciones del *script* solo se llevarán a cabo en la instancia local a la máquina del jugador, y se sincronizará la posición en las demás instancias gracias al NetworkTransform. Podemos saber si la instancia que está ejecutando es la del jugador local gracias al atributo `isLocalPlayer` de la clase `NetworkBehaviour`. Por tanto, las instrucciones de inicialización, actualización y reacción a otros objetos solo se llevarán a cabo si `isLocalPlayer` es verdadero.
- Como el jugador es instanciado por el NetworkManager, será preciso especificar a la cámara virtual Cinemachine cuál es el objeto a seguir en tiempo de ejecución. Por ello, añadiremos la etiqueta “CameraSet” a la cámara virtual y en el método `Start()` del jugador local la utilizaremos para encontrar la cámara virtual y modificar su campo `Follow` al componente `Transform` del avatar del jugador.

Para que el jugador pueda tomar control del avatar, será necesario convertir el objeto player en un *prefab* y eliminarlo de la escena. Esto se debe a que será el NetworkManager el responsable de la instanciación del avatar del jugador. Para ello, añadiremos el *prefab* del jugador al campo *Player Prefab* en la sección *Spawn Info* del componente NetworkManager.

Con estos pasos, ya podemos proceder a la conexión de un cliente y un host haciendo uso de la interfaz proporcionada por NetworkManagerHUD. Cuando se complete la conexión, veremos a los dos jugadores en pantalla, como puede verse en la Figura 20. Cada uno podrá moverse y sus movimientos se reflejarán en la otra instancia. El tercer paso será implementar la lógica del inicio de partida.

El objeto GameStarter deberá contener gracias al nuevo *script MatchStarter.cs* el número máximo de jugadores permitidos en una carrera, el número mínimo de jugadores necesarios, el número actual de jugadores registrados para la carrera, el tiempo total en segundos que durará la cuenta atrás antes de iniciar la carrera y finalmente el valor actual de la cuenta atrás. También será necesario disponer de un atributo booleano que indique si ya se ha solicitado el acceso a una partida. De esta manera, cuando el jugador local interactúe con GameStarter mediante el uso de `Interact()` se registrará la participación del mismo en la carrera siempre que no se haya alcanzado el número máximo de participantes ni se haya registrado ya su participación. Para poder sincronizar el número de participantes entre todas las instancias cuando un jugador se registra para la carrera, el primer paso necesario es

que la instancia host incremente en una unidad el número de jugadores actualmente registrados y, posteriormente, se procederá a la actualización de este valor en todos los clientes. Para poder conseguir esta sincronización, necesitamos:

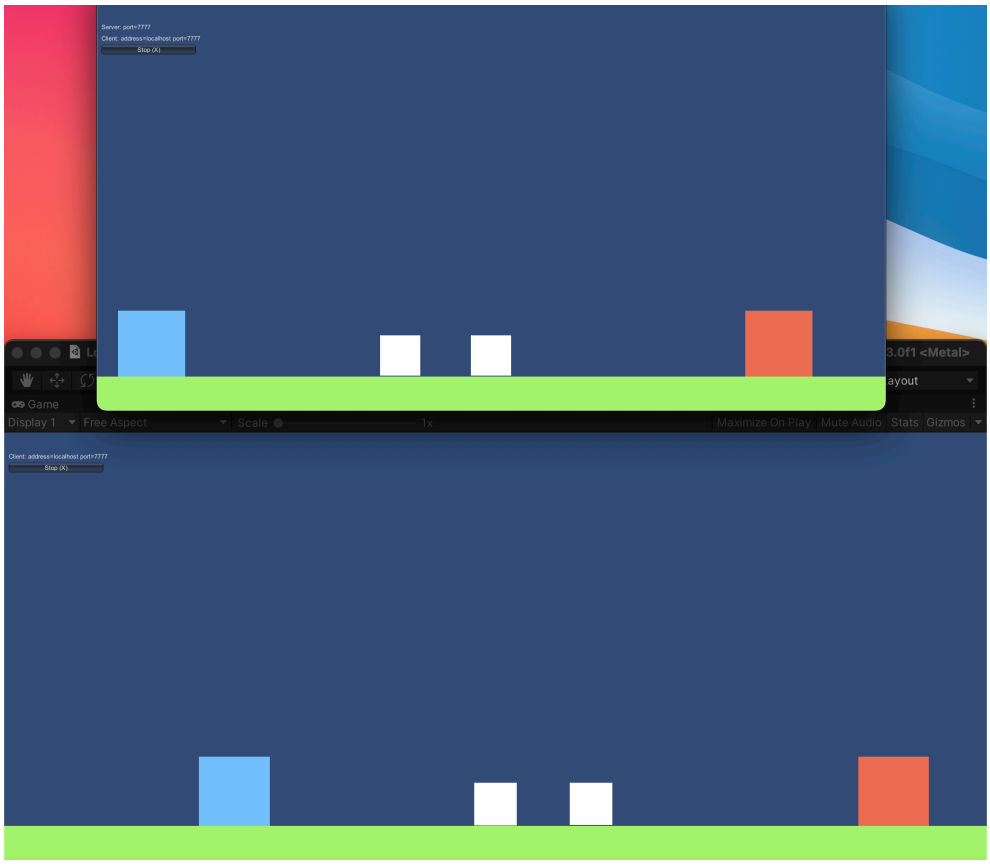

Figura 20: conexión sencilla entre un host y un cliente.

- Añadir el atributo `[SyncVar]` al campo `currNumberOfPlayers`. Este atributo proporcionado por UNet sirve como indicador de que esta variable debe ser sincronizada entre todas las instancias del objeto de forma automática. Cabe destacar que este atributo implica que la variable únicamente podrá ser modificada por la instancia situada en el host, aunque con la ventaja de que garantiza que, al conectarse un nuevo jugador, tendrá el valor de la variable actualizado desde el comienzo.

- Para poder pedirle a la instancia servidor que incremente el número actual de jugadores es necesario hacer uso de RPC dirigidas del cliente al servidor, lo que en HLAPI se conocen con el nombre de comandos. Para definir una función como comando, será necesario añadirle el atributo `[Command]` y que el nombre de la función comience por el prefijo "Cmd..." [48]. Así, definimos el comando `CmdUpdateNumberOfReadyPlayers()` que incrementará el número de jugadores registrados en una unidad. Es importante tener en mente que los comandos se ejecutarán en el contexto de la instancia servidor, y no en el contexto de la instancia que lo llama.

El siguiente paso será proceder a iniciar la cuenta atrás cuando el número actual de jugadores registrados supere el número mínimo de jugadores necesarios. La lógica de dicha cuenta atrás deberá realizarse únicamente si estamos en la instancia del host, es decir, cuando la variable `isSever` de `NetworkBehaviour` sea verdadera. Cuando la

cuenta atrás se inicie, se deberá informar a todos los jugadores registrados, por lo que deberá sincronizarse dicha información con un booleano entre todas las instancias de GameStarter. Para llevar a cabo esta sincronización, haremos uso de una RPC dirigida del servidor a todos los clientes, la cual basta con definir mediante el uso del atributo `[ClientRPC]` delante de la función y que el nombre de la misma contenga el prefijo "Rpc…" [48]. De esta manera, definimos `RpcGetReadyForMatch()` que actualizará la información del inicio de la cuenta atrás en los clientes. Cuando la cuenta atrás empiece, la instancia servidor se encargará de actualizar su valor, el cual se sincronizará para todos los jugadores que vayan a participar en la carrera mediante `RpcUpdateCountdown()`. Finalmente, cuando la cuenta atrás llegue a cero, deberá procederse a la partición de los jugadores según vayan a correr o no y la carga de la escena de juego para los que vayan a correr. La lógica relativa a esta administración de jugadores se llevará a cabo en una clase a parte especializada en dicha tarea, la cual detallaremos más adelante. Por tanto, el papel de GameStarter en dicho proceso será la llamada en la instancia del host de `RpcSendSplit()`, la cual provocará que cada uno de los clientes emita un mensaje local a través de `Messenger` asociado al evento `NetworkEvent.SPLIT`. Hay que tener en cuenta a la hora de diseñar las RPC y los comandos que el dispositivo que actúe como host, contendrá a su vez de manera interna un cliente y un servidor. Esto significa que al ejecutar una RPC en el host, este también se ejecutará de forma interna en la parte cliente del mismo.

Con esto ya tendríamos una primera versión del código *MatchStarter.cs* necesario para GameStarter. Para que GameStarter realmente pueda sincronizarse a través de la red, será preciso añadirle el componente NetworkIdentity con la opción *Local Player Authority* activada. Esta opción es necesaria para permitir la ejecución de comandos en el servidor por parte de los clientes. Si esta opción no se activa, el objeto se instancia por defecto en modo *Server Authority*, por el cual únicamente la instancia servidor puede llevar a cabo acciones remotas [45].

No obstante, en realidad la versión actual de *MatchStarter.cs* no funciona como cabría esperar, ya que los comandos no se ejecutan en la instancia servidor. Un ejemplo de este comportamiento inesperado puede verse al tratar de interactuar con GameStarter en un cliente. En este caso, la instancia cliente trata de invocar el comando `CmdUpdateNumberOfReadyPlayers()`, pero este no se ejecuta en la instancia servidor y por tanto no se incrementa el número de jugadores listos para correr. Esto se debe a que, a pesar de haber activado la opción *Local Player Authority*, realmente las instancias cliente del GameStarter no tienen la autoridad necesaria para alterar la instancia servidor. Esto es debido a una medida de seguridad impuesta por Unity en el diseño de HLAPI para evitar que objetos que no son el jugador puedan verse alterados de forma local en una máquina y que su estado se sincronice en todos los dispositivos, previniendo así posibles trampas. Por tanto, la forma para que un objeto no-jugador pueda ejecutar comandos es a través de la propiedad (*ownership*). Gracias al método `AssignClientAuthority(…)` de la clase `NetworkIdentity` podemos dar a un objeto no-jugador los permisos necesarios para poder ejecutar comandos, ya que esta función hará que el objeto pase a tener como propietario al jugador local [45]. La Figura 21 muestra el esquema del proceso de ejecución de comandos en un objeto no-jugador.

Por tanto, será necesario antes de poder ejecutar un comando en GameStarter solicitar la autoridad necesaria al jugador. Para ello:

- Modificamos *Player.cs* para añadir el comando público `CmdSetAuth(…)` [41]. Dicho comando se encargará de comprobar quién es el propietario del objeto que solicita la autoridad gracias a `NetworkIdentity.clientAuthorityOwner`. Si el propietario no es el jugador local, primero se elimina el vínculo con el propietario actual mediante el método `RemoveClientAuthority(…)` de `NetworkIdentity` y a continuación se vincula con el jugador local gracias a `AssignClientAuthority(…)`. La necesidad de implementar este comando en el avatar del jugador se debe a que los objetos jugador son los únicos que tienen la autoridad necesaria para ejecutar el comando que permite que el servidor modifique el nivel de autoridad del objeto no-jugador. Finalmente, modificaremos la función `Start()` para que, cuando la instancia sea el jugador local, haga uso de la etiqueta "LocalPlayer".

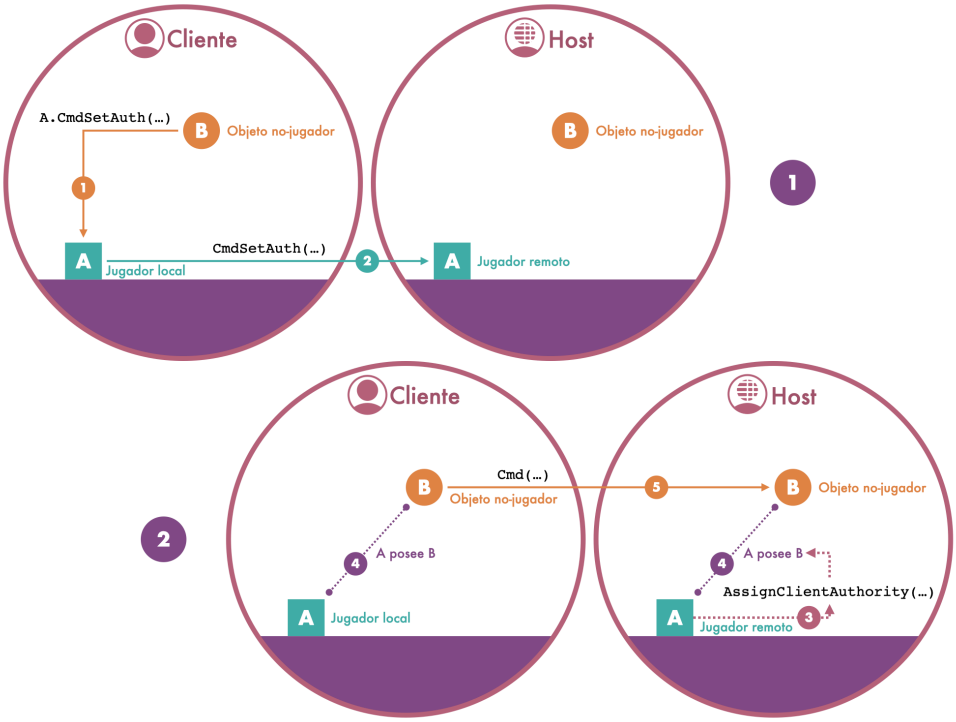

Figura 21: proceso de solicitud de autoridad para la ejecución de un comando en un objeto no-jugador.

- Debido a que la ejecución de comandos por parte de objetos no-jugadores será una práctica común, para evitar la sobrecarga de las tareas de la clase `Player` y facilitar la modularidad, es necesaria la implementación de una nueva clase que incorpore las funciones que permitan la solicitud de la autoridad y la ejecución de comandos. Por tanto, implementamos la clase `DistributedEntityBehaviour` con este fin. Esta clase hereda de `NetworkBehaviour` e implementa las funciones:

    - `GetAuthority()`: corrutina encargada de solicitar al jugador la autoridad para ejecutar comandos mediante la búsqueda del jugador local gracias a la etiqueta "LocalPlayer" y la llamada al comando del jugador `CmdSetAuth(…)`.

    - `RemoveAuthority()` / `RemoveOwnership()`: funciones encargadas de eliminar la autoridad adquirida haciendo uso de la función

`RemoveClientAuthority(…)`. `RemoveOwnership()` es una alias para `RemoveAuthority()`.

- `RunCommand(CommandNoArguments cmd)`: corrutina encargada de la ejecución de un comando sin argumentos. Para ello la corrutina solicita la autoridad necesaria mediante `GetAuthority()` y cuando la recibe ejecuta la función `cmd`.

- `RunCommand<T>(CommandOneArgument<T> cmd, T arg)`: igual que `RunCommand(…)`, pero para comandos que requieran un argumento `arg` en su llamada.

- `RunCommand<T, D>(CommandTwoArguments<T, D> cmd, T arg1, D arg2)`: igual que `RunCommand(…)`, pero para comandos que requieran dos argumentos `arg1` y `arg2` en su llamada.

- Finalmente, `MatchStarter` heredará de `DistributedEntityBehaviour` en lugar de `NetworkBehaviour`. De esta manera, cada vez que queramos ejecutar un comando, bastará con usar `RunCommand(…)`. No obstante, es necesario hacer unas puntualizaciones:

  - No se puede pasar directamente el comando a ejecutar como argumento de `RunCommand(…)`, ya que si procedemos de esta manera el comando tratará de ejecutarse en el cliente en lugar de en el servidor. Para poder ejecutarlo correctamente, será necesario encapsular la llamada al comando dentro de otro método, que será el que pasaremos a `RunCommand(…)`.

  - En la implementación de los comandos, habrá que poner como última instrucción la llamada a `RemoveAuthority()`. Esto no es obligatorio, pero si se desea evitar destrucciones indeseadas del objeto es recomendable, ya que en la concesión de la autoridad se vincula el objeto como una pertenencia del jugador local, con lo que si el jugador local es destruido por algún motivo, como por ejemplo una desconexión, el objeto vinculado también se destruirá para todos los jugadores.

  - La solicitud de la autoridad no asigna autoridad a todo el objeto, sino a la instancia concreta que la ha pedido. Esto se debe a que se vincula como propietario del objeto al jugador local, que es el que tiene autoridad en la máquina (A) en la que se ha solicitado. No obstante, el jugador de A no tiene autoridad en ninguna otra máquina (B), con lo que la instancia del objeto en B tampoco tendrá autoridad. Esto fuerza a tener que comprobar si se tiene la autoridad y pedirla si no se posee antes de ejecutar cualquier comando.

Con todo esto ya tenemos la funcionalidad necesaria para el objeto GameStarter. Solo nos falta añadir la emisión de mensajes locales asociados a los eventos `LobbyEvent.WAITING_FOR_MATCH` y `LobbyEvent.MATCH_COUNTDOWN_UPDATE` en los métodos `Interact()` y `RpcUpdateCountdown()`.

A continuación será necesario implementar la funcionalidad relativa a la administración de los grupos de juego, de *lobby* y la separación de los mismos. Para ello crearemos un nuevo objeto vacío en la escena llamado PlayersManager que se

ocupará de estas tareas. Este objeto necesita un componente NetworkIdentity con la propiedad *Local Player Authority* activada, ya que será necesario que las instancias cliente modifiquen la instancia servidor para poder llevar a cabo la administración de los grupos. Finalmente, crearemos un nuevo *script* llamado *PlayersManger.cs* que implementará la funcionalidad necesaria.

La clase `PlayersManager` hereda de `DistributedEntityBehaviour` y permitirá la persistencia del objeto entre escenas gracias al uso en el método `Awake()` de `DontDestroyOnLoad(gameObject)`. Cabe destacar de esta clase los campos:

- `myPlayerId` que almacena el identificador del jugador local a la instancia.
- `numberOfPlayers` que almacena el número de jugadores registrados. Se empleará el atributo `[SyncVar]` para su sincronización.
- `currentPlayerGroup` que contiene la lista de identificadores de jugadores en el grupo del *lobby*.
- `nextPlayerGroup` que contiene la lista de identificadores de jugadores en el grupo de juego.
- `playersIpAddresses` que contiene un diccionario con las direcciones IP de los clientes asociadas con el identificador de su jugador local. De esta manera, a partir del identificador se puede obtener la dirección IP del cliente, lo que será necesario para la redirección de los clientes al crear un nuevo host.

Así, la clase empieza por registrar a los jugadores en el grupo de *lobby* al comenzar. Este proceso debe llevarse a cabo cuando se establece una conexión con un host, lo que es posible gracias a la función `OnStartClient()` de `NetworkBehaviour` que se ejecutará cuando se inicie un cliente (en el caso del host, también se llamará cuando se inicie el cliente interno). En dicha función iniciaremos una corrutina que buscará al jugador local a través de la etiqueta "LocalPlayer" y obtendrá su identificador gracias al campo `netId` de su NetworkIdentity, almacenándolo en `myPlayerId`. Tras obtener el identificador, si la instancia se está ejecutando en un cliente, se llamará al comando `CmdAddPlayer(…)` haciendo uso de `RunCommand(…)` y proporcionando como argumento el identificador del jugador a añadir. Gracias a este comando, la instancia del servidor añadirá el identificador a la lista `currentPlayerGroup`. También se registrará la dirección IP asociada a este cliente gracias a que la función `FindLocalObject(…)` de `NetworkServer` nos permite encontrar el jugador a partir del id proporcionado y obtener la dirección IP al consultar el campo `connectionToClient.address` de su `NetworkIdentity`. Si la instancia se está ejecutando en el servidor, el identificador se puede añadir directamente.

El seguimiento de los mensajes locales asociados a `NetworkEvent.SPLIT` y `LobbyEvent.WAITING_FOR_MATCH` permite llevar a cabo el proceso de separación de los jugadores en la clase `PlayersManager`. El primero causará la ejecución del método `RunSplit()`, cuando la instancia esté en el servidor, mientras que el segundo permitirá la ejecución de `CmdPreparePlayerSplit(uint nwId)`.

Por un lado, `CmdPreparePlayerSplit(uint nwId)` es el método responsable de la primera fase en la separación de los jugadores. En esta fase, se extraen los jugadores

que vayan a participar en la carrera del grupo de *lobby* y se insertan en el grupo de juego. Así este método eliminará `nwId` de la lista `currentPlayerGroup` y lo añadirá a la lista `nextPlayerGroup`.

Por otra parte, `RunSplit()` se encargará de la segunda fase, es decir, del inicio de la división de los jugadores. Como hemos definido en el apartado de diseño, el procedimiento de separación a seguir dependerá de si el host se encuentra en el grupo de *lobby* o en el grupo de juego:

- Si `nextPlayerGroup` contiene el `myPlayerId` de la instancia servidor significa que el host va a jugar. Por tanto, nombraremos al primer jugador de `currentPlayerGroup` el nuevo host mediante la RPC `RpcBecomeHost(…)` y para el resto de jugadores de la lista se les indicará el cambio de conexión al nuevo host gracias a la RPC `RpcChangeClientConnection(…)`. Finalmente, se procede al cambio de escena para todos los jugadores restantes gracias al método `ServerChangeScene(…)` de `NetworkManager`, para lo que será necesario disponer de una referencia al objeto NetworkManager.
- Si no, se nombrará como nuevo host al primer jugador de `nextPlayerGroup`, indicándole que proceda a cambiar de escena cuando todos los participantes se conecten a través de `RpcBecomeHost(…)`. Al resto de jugadores de la lista, se les pedirá que procedan a cambiar su conexión al nuevo host mediante `RpcChangeClientConnection(…)`.

El método `RpcBecomeHost(uint nwId, int numOfPly, int port, bool changeScene, string scene)` únicamente ejecutará la conversión a host de un cliente si `nwId` es igual `myPlayerId`. En este caso, se procede a interrumpir la conexión con el host actual mediante `StopClient()` de `NetworkManager`. A continuación, se procede a iniciar una nueva conexión como host en el puerto `port` mediante el método `StartHost()` de `NetworkManager` previamente modificando el campo `networkPort` del mismo. Finalmente, si es `changeScene` verdadero, se procede al inicio de una corrutina que esperará hasta que el número de jugadores conectados sea igual a `numOfPly` para llevar a cabo el cambio a la escena `scene` mediante `ServerChangeScene(…)`.

El método `RpcChangeClientConnection(uint nwId, int port, string address)` únicamente ejecutará la reconexión a un nuevo host si `nwId` es igual `myPlayerId`. En este caso, se procede a interrumpir la conexión con el host actual mediante `StopClient()` de `NetworkManager`. A continuación, se procede a iniciar una nueva conexión con el host situado en `address` a través del puerto `port` haciendo uso del método `StartClient()` de `NetworkManager`, previamente modificado el campo `networkPort` y `networkAddress` del mismo.

Con esto ya tenemos implementada la funcionalidad necesaria para administrar la separación de los jugadores y el inicio de la carrera. No obstante, cabe destacar que en la implementación actual de la HLAPI de Unity, el funcionamiento de `PlayersManager` no es el esperado. Esto se debe a la función `OnStarClient()`. En su definición, `OnStartClient()` debe ser llamada de forma automática cada vez que se establece una nueva conexión a un host [51]. Si probamos a ejecutar dos instancias del sistema construido hasta el momento, un host y un cliente, y procedemos a la

conexión del cliente, veremos que `OnStartClient()` se llama y por tanto se registra el jugador local del cliente en la lista `currentPlayerGroup` del host. No obstante, si procedemos a desconectar el cliente y a volverlo a conectar, veremos que el jugador local no se registra en la lista `currentPlayerGroup` del host.

Esto se debe a que `OnStartClient()` no se ejecuta al producirse una reconexión. Tras proceder a la búsqueda de este fenómeno en la red, encontramos que el mismo había sido reportado y confirmado por Unity. No obstante, debido al fin del soporte de UNet y al no considerarse un error crítico, Unity reportó que no iba a ser reparado [54]. Por suerte, gracias al usuario Goldbug y sus comentarios en la página donde se reportaba el error [54], fue posible acceder a una solución al problema [61]. Dicha solución, la cual puede verse en la Figura 22, consiste en la modificación de la clase `ClientScene` de la HLAPI de UNet para incorporar tres instrucciones adicionales con el fin de garantizar el correcto reinicio de la misma. Para llevar a cabo estas modificaciones, fue necesaria la modificación del código original de la HLAPI de UNet. Esta modificación requirió la incorporación directa de los archivos de HLAPI obtenidos de la caché de Unity a la carpeta *Packages* del proyecto, eliminar la dependencia con Multiplayer HLAPI de *manifest.json* y finalmente modificar el código *ClientScene.cs* según las indicaciones. Con estas modificaciones ya podemos proceder a la adaptación de la escena de juego, tras lo cual retornaremos a finalizar la escena del *lobby*.

Figura 22: captura de la solución proporcionada por el usuario Goldbug [61] para el error relativo al mal funcionamiento de `OnStartClient()` [54].

5.3.3.3.Carrera

Ahora que ya hemos implementado la separación de los clientes y la carga de la escena de juego, podemos proceder a la adaptación de la misma. En primer lugar, al igual que hemos procedido en el caso del *lobby*, vamos a analizar qué objetos será necesario que sean sincronizados a través de la red y cuáles permanecerán como objetos locales:

- El avatar del jugador será el mismo que en el *lobby*, con lo que este objeto se reutilizará.
- La generación del terreno se debe llevar a cabo en el host y ser sincronizada en todos los clientes, de forma que todos los jugadores tengan el mismo terreno. Por ello, el objeto CellGenerator, así como las celdas generadas, deberán ser objetos en red.

- La interfaz muestra la información relativa al jugador local y no requiere ningún tipo de sincronización con las demás instancias, por lo que constituirá un objeto local.
- La cámara y la cámara virtual Cinemachine, como en el *lobby*, son objetos locales ya que cada instancia deberá seguir a su jugador local y será independiente de las demás.
- La pared trasera, como en la versión local, se encargará de impedir que los jugadores puedan retroceder hacia la izquierda, permitiendo así eliminar el terreno ya superado y clarificando al usuario el sentido de avance. Por tanto, este objeto avanzará según la coordenada X del jugador en la última posición, y su posición deberá ser la misma para todas las instancias, por lo que será un objeto en red.
- El objeto GameController no tiene necesidades de sincronización entre las diferentes instancias. No obstante, la instancia servidor de GameController contará con funcionalidad adicional ya que se encargará de determinar las posiciones del primer y último jugador. Por tanto, para acceder a la propiedad `isServer` y adaptar su funcionalidad según el tipo de instancia en la que se encuentre necesitará ser un objeto en red.

La escena de juego no necesitará la adición de un NetworkManager, ya que el comportamiento de este objeto es persistente por defecto, de forma que no será destruido al cambiar de escena. Por tanto, como el jugador siempre deberá pasar por la escena de *lobby* antes de la carrera, el NetworkManager de dicha escena seguirá activo al entrar en la escena de juego. Por este motivo, tampoco será necesario tener el avatar del jugador en la escena, ya que el NetworkManager se encargará de su creación al cambiar de escena. Por tanto, procederemos a la adaptación de CellGenerator y `CourseGenerator`.

Para este objeto será necesaria la incorporación del componente NetworkIdentity como para todos los objetos en red, pero también será preciso agregar el componente NetworkTransform en modo *Sync Transform* como modo de sincronización ya que CellGenerator no posee un RigidBody2D. Esto se debe a que CellGenerator se desplaza con cada nueva columna generada, y será necesario sincronizar dicho desplazamiento en cada cliente. En cuanto al código *CourseGenerator.cs* que implementa la generación procedural del terreno, será preciso realizar las modificaciones necesarias para asegurar que la generación solo se produce en la instancia servidor gracias a `isServer`, por lo que la clase `CourseGenerator` heredará de `NetworkBehaviour`.

Otra modificación importante a llevar a cabo es la forma en que las nuevas celdas generadas se instancian. Unity, en el caso de los juegos locales, permite la creación de nuevos objetos en la escena a partir de *prefabs* gracias al método `Instantiate(…)`. No obstante, en escenas en red este método únicamente provoca la instanciación del objeto de forma local. Para poder instanciar dicho objeto en todos los clientes, es necesario pedirle al servidor que lleve a cabo dicha instanciación, para lo que hay que llamar al método `Spawn(…)` de `NetworkServer`. Así, el proceso es el siguiente:

1. Se instancia el `prefab` deseado de forma local en el host mediante `Instantiate(prefab)`. Se guarda el objeto devuelto en una variable (`objt`).
2. Se llevan a cabo las operaciones necesarias sobre el objeto de dicha variable.

3. Se instancia el objeto `objt` en todos los clientes mediante la instrucción `NetworkServer.Spawn(objt)`.

Además, es necesario llevar a cabo una configuración adicional para permitir la instanciación de objetos en todos los clientes. `NetworkServer` solo será capaz de instanciar *prefabs* que hayan sido previamente registrados en el NetworkManager. Estos *prefabs* instanciables pueden añadirse desde el menú del componente NetworkManager en el campo *Registered Spawnable Prefabs*. Por tanto, en nuestro caso será preciso modificar el NetworkManager del *lobby* para registrar los *prefabs* asociados a las celdas que se pueden generar. Un requisito necesario para poder llevar a cabo este registro será que el *prefab* que se quiera registrar sea un objeto en red y contenga, por tanto, el componente NetworkIdentity [49]. En consecuencia, será preciso también modificar los *prefabs* de las celdas para añadirles el componente NetworkIdentity.

El último cambio necesario a realizar en la clase `CourseGenerator` será la suscripción a los mensajes locales asociados al evento `GameEvent.FIRST_PLAYER_POSITION_CHANGED` que transmitirán la coordenada X de la posición del primer jugador. Modificaremos el código de `Update()` para llevar a cabo la generación en función de este valor en lugar de la posición de la cámara. Con estos cambios la generación del terreno se sincronizará de forma adecuada en todos los clientes.

El siguiente paso a realizar será la modificación del objeto Floor, al que renombraremos como BackBarrier, ya que limita el recorrido hacia atrás del jugador. Para ello, le añadiremos los componentes NetworkIdentity y NetworkTransform en modo de sincronización *Sync Transform* para que todas las instancias del objeto tengan la misma posición. A continuación, modificamos el *script ForwardHorizontalDisplacement.cs* para que la clase herede de `NetworkBehaviour`. Como su posición dependerá de la posición del último jugador, se modifica el código para que en lugar de seguir a un objeto dado `target`, se registren los mensajes locales asociados a los eventos `GameEvent.PLAYER_STARTS` y `GameEvent.LAST_PLAYER_POSITION_CHANGED` por parte de la instancia del servidor. Estos eventos nos permitirán conocer la coordenada X de la posición del último jugador y actualizar nuestra posición de forma correspondiente. De nuevo, el cambio de la posición solo es llevado a cabo por la instancia servidor y se sincroniza en los clientes a través del componente NetworkTransform.

Un problema que surge de estos cambios reside en el confinador empleado para la cámara virtual Cinemachine. Dicho confinador consiste en un PolygonCollider2D que se encuentra vinculado al objeto BackBarrier y que asegura que la cámara no se saldrá de sus límites. No obstante, debido a que en la versión en red la posición de BackBarrier depende del último jugador, esto provocará que las cámaras de los jugadores más avanzados queden atrapadas por el *confiner* sin poder seguir al jugador local. Para poder solucionar este problema, sería necesario definir el área accesible para la cámara en tiempo de ejecución como el área rectangular comprendida entre la pared trasera y el objeto CellGenerator. Para conseguir este comportamiento diseñamos una extensión para Cinemachine [6] llamada *Dynamic Delimiter 2D*. Dicha extensión recibe cuatro `Transform` y adapta la posición de la cámara en tiempo de ejecución para que no se salga del área rectangular definida por dichos `Transform`. De esta

manera, basta con emplear el objeto Wall como límite izquierdo y el objeto CellGenerator como límite derecho para limitar el movimiento horizontal de la cámara. Dos nuevos objetos vacíos se emplearán para definir los límites superior e inferior de forma constante. De esta manera, el tamaño horizontal del área accesible por la cámara cambiará según la distancia entre Wall y CellGenerator evitando que la cámara quede atrapada como puede verse en la Figura 23.

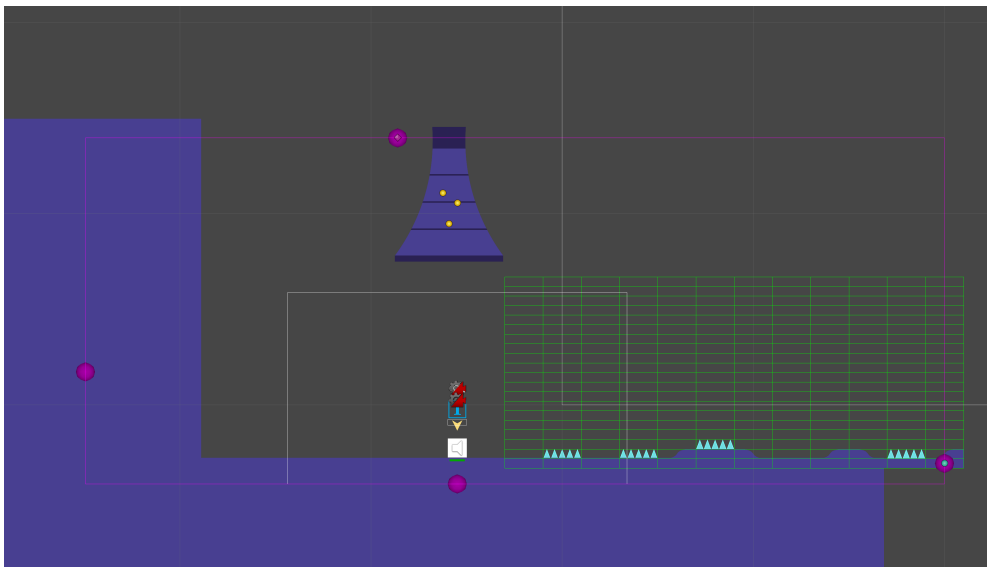

Figura 23: visualización del área delimitada gracias a *Dynamic Delimiter 2D* en color magenta.

A continuación, procederemos a la adaptación de GameController. Como en los casos anteriores, le añadiremos el componente NetworkIdentity y llevaremos a cabo las modificaciones necesarias en la clase `GameController`, que también pasará a heredar de la clase `NetworkBehaviour` en lugar de `MonoBehaviour`. Así, extenderemos la funcionalidad de `GameController` para:

- Llevar a cabo la actualización de la posición de los jugadores en primera y última posición. Para ello, obtendrá todos los objetos con la etiqueta "Player" (asignada al *prefab* del avatar del jugador) y comparará su posición en la coordenada X con la posición en dicha coordenada del objeto con la etiqueta "LocalPlayer" si el jugador local aún está vivo. De esta manera se determina la mínima y máxima posición de entre todos los jugadores, y se comunica mediante la emisión de mensajes locales asociados a los eventos `GameEvent.LAST_PLAYER_POSITION_CHANGED` y `GameEvent.FIRST_PLAYER_POSITION_CHANGED` respectivamente. Esta funcionalidad únicamente se llevará a cabo en la instancia servidor.

- Cuando el jugador emita el mensaje local `GameEvent.PLAYER_DIED`, la clase se encargará de controlar la cámara virtual para modificar el campo `Follow` de la misma. De esta manera, dependiendo de la entrada proveniente de las flechas horizontales del teclado y obteniendo el conjunto de objetos con la etiqueta "Player", se puede modificar el jugador visto en modo espectador.

- Por último, cuando la salud del jugador local llegue a cero y la talla del conjunto de objetos con la etiqueta "Player" sea cero, se emitirá un mensaje local asociado al evento `GameEvent.GAME_FINISHED`, ya que la carrera habrá terminado.

Finalmente, adaptaremos los elementos relativos a la interfaz. Para ello es necesario modificar *CourseUI.cs* para que en lugar de mostrar la pantalla de final de partida al morir el jugador, se muestre cuando todos los jugadores hayan terminado. Por tanto modificamos la clase `CourseUI` para que haga seguimiento de los mensajes locales asociados al evento `GameEvent.GAME_FINISHED`, de tal forma que al recibir un mensaje asociado a `GameEvent.PLAYER_DIED` se ocultará la barra de salud y el contador de distancia de la interfaz, mientras que cuando se reciba `GameEvent.GAME_FINISHED` se mostrará la pantalla de fin. A continuación, procederemos a la modificación de *ScoreScreen.cs* para que, antes de proceder al cambio de escena cuando el jugador presione la tecla intro, se lleve a cabo la destrucción del NetworkManager, así como de otros objetos con la propiedad `DontDestroyOnLoad`. Esto se hace para garantizar que no se producirán conflictos con los objetos de gestión de red en el *lobby*. La transición de la escena se llevará a cabo de forma local con `SceneManager.LoadScene("LobbyScene")`. Cabe destacar que la destrucción del objeto NetworkManager debe ser seguida por:

- El reinicio de la escena en red actual mediante `NetworkManager.networkSceneName = ""`. Esto es necesario ya que si no lo cambiamos manualmente, cuando el cliente proceda a la reconexión con un host al entrar al *lobby*, tratará de volver a cargar la escena del juego ya que `NetworkManager.networkSceneName` sigue teniendo el valor "GameScene".

- El apagado completo del sistema NetworkManager para asegurar su correcto reinicio en la carga del *lobby*. Esto se consigue gracias a `NetworkManager.Shutdown()`.

Con esto ya tendríamos la adaptación de los componentes de la escena de juego. No obstante, aún nos falta implementar la correcta desconexión de los clientes y servidores al terminar la carrera. Para ello, añadiremos la funcionalidad necesaria a la clase `PlayersManager`, ya que es la responsable de administrar las conexiones y, gracias a la propiedad `DontDestroyOnLoad`, persistirá a los cambios de escena con lo que estará disponible en la escena de juego. Las adiciones a realizar sobre la clase `PlayersManager` son:

- Incorporación de la corrutina `CloseHost()`: esta corrutina se mantendrá en espera mientras aún haya clientes conectados y procederá a la desconexión del host tras la espera mediante el método `StopHost()` de `NetworkManager`. Finalmente, el objeto se autodestruirá gracias a `Destroy(gameObject)`.

- Seguimiento de los mensajes asociados al evento `GameEvent.FINISHED_SCREEN_IS_OUT` gracias a la *callback* `OnFinishedScreenOut()`. Este método procederá a la ejecución de `CloseHost()` cuando se invoque en la instancia servidor, mientras que ejecutará la función `StopClient()` de NetworkManager y procederá a su autodestrucción cuando esté en una instancia cliente.

De esta manera, mediante la emisión del mensaje local asociado a `GameEvent.FINISHED_SCREEN_IS_OUT` por parte de `CourseUI` cuando la carrera termine, se procederá a la desconexión de los clientes y, en última instancia, del host. Así conseguimos una correcta reconexión en el *lobby*.

De vuelta en el *lobby*, aún hay aspectos que requieren nuestra atención. Por un lado, como habíamos establecido en nuestro diseño, queremos que la conexión entre los jugadores se produzca de forma orgánica y transparente, sin tener que recurrir a la introducción manual de la dirección IP y el puerto del host. Esto implicará el uso de un sistema de descubrimiento de jugadores. Por otro lado, también será necesario la creación de un nuevo *script* para el objeto LobbyExit que lleve a cabo la correcta desconexión antes de regresar a la pantalla del menú.

En primer lugar, abordaremos la implementación del sistema de descubrimiento. Este se basará en el uso de la clase `NetworkDiscovery` de HLAPI. La clase `NetworkDiscovery` permite el descubrimiento de jugadores en la red local, no siendo necesario el conocimiento previo de la dirección IP del host [46]. Para utilizar este sistema, creamos un nuevo objeto vacío al que llamaremos PlayerFinder. Este objeto incorporará el componente NetworkDiscovery y un nuevo *script* llamado *PlayersFinder.cs*. Así, la clase `PlayerFinder` se encargará del control del componente NetworkDiscovery. Antes de explicar la implementación de esta clase, cabe destacar las siguientes funciones de la clase `NetworkDiscovery`:

- `Initialize()`: inicia el sistema de descubrimiento comprobando que se puede hacer uso del puerto de emisión/recepción de mensajes de descubrimiento.
- `StartAsClient()`: activa el sistema de descubrimiento en modo cliente, escuchando los mensajes provenientes de hosts activos en la red local.
- `StartAsSever()`: activa el sistema de descubrimiento en modo servidor, enviando los mensajes de descubrimiento. El contenido en formato de cadena de dichos mensajes es de la forma "NetworkManager:*dirección*:*puerto*" al activar en el componente la opción *Use Network Manager*. Esta cadena se puede obtener a partir del mensaje mediante la modificación de `NetworkDiscovery` haciendo público su método `BytesToString(…)`.
- `StopBroadcast()`: detiene la emisión y escucha de mensajes de descubrimiento.

Haciendo uso de estos métodos, la clase `PlayersFinder` se encarga de iniciar el componente en modo cliente al comenzar la ejecución. Tras un tiempo de espera aleatorio, si el sistema no ha recibido ningún mensaje de un host, se detendrá la escucha de los mensajes. Se iniciará entonces la emisión de mensajes como servidor, iniciando la máquina actual también como host gracias a `StartHost()` de `NetworkManager`. Por el contrario, si el sistema recibe una invitación, procederá a establecer la conexión con el host que la envió. Para ello, el código comprueba en cada actualización de `Update()` el campo `broadcastsReceived` del `NetworkDiscovery`, el cual contiene todos los mensajes recibidos. Si su talla es mayor que cero, se obtiene la primera invitación recibida y se procede a la obtención de la dirección y el puerto a partir de ella. La dirección puede encontrarse directamente en el campo `serverAddress` de la clase `NetworkBroadcastResult`, mientras que para la obtención del puerto será necesaria la conversión de los *bytes* del mensaje en una cadena. Una vez obtenida la cadena, podemos obtener el entero correspondiente al puerto teniendo en cuenta su formato e iniciar así un cliente mediante `StartClient()` tras modificar los campos `networkAddress` y `networkPort` del `NetworkManager`. Finalmente detendremos la recepción de mensajes de descubrimiento.

En segundo lugar, para poder llevar a cabo la desconexión de los jugadores de forma apropiada incorporaremos dos nuevos métodos públicos a la clase `PlayersManager`:

- `IsolateClient(…)`: inicia la corrutina `Unregister(…)`, la cual permitirá la ejecución del comando `CmdUnregister(…)`. Este comando eliminará el identificador del jugador local del cliente de la lista correspondiente en la instancia servidor y su dirección IP de `playersIpAddresses`. Una vez se elimine la autoridad de la instancia, procederá a la desconexión del cliente mediante `StopClient()` y su autodestrucción.

- `IsolateHost(…)`: escogerá el primer jugador disponible y lo convertirá en el nuevo host mediante `RpcBecomeHost(…)`. Luego conectará cada uno de los clientes restantes con el nuevo host mediante `RpcChangeClientConnection(…)`. Finalmente, detendrá la emisión de mensajes de descubrimiento por parte de `PlayersFinder`. Cuando todos los clientes se hayan desconectado, precederá a la desconexión del host mediante `StopHost()` y a la autodestrucción del objeto.

Así, la nueva clase `LobbyCloser` para LobbyExit hará uso de estos métodos. Esta clase hereda de `NetworkBehaviour` e implementa la interfaz `InteractableObject`, de tal forma que cuando se ejecuta su método `Interact()`, se procede a la búsqueda del objeto PlayersManager gracias a la etiqueta "PlayersManager". Una vez encontrado, se invocará al método `IsolateHost(…)`, en el caso de la instancia cliente, o al método `IsolateClient(…)`, en el caso de la instancia servidor. Ambos métodos tras el proceso de desconexión invocarán el método público `Close()` de `LobbyCloser`, el cual llevará a cabo la destrucción del NetworkManager, su apagado mediante `NetworkManager.Shutdown()` y el reseteo de la escena en red. Finalmente, se cargará el menú principal de forma local mediante `SceneManager.LoadScene(…)`.

5.3.3.4.Finalización del lobby

El último paso a realizar para finalizar la adaptación de la escena del *lobby* es la incorporación de una interfaz que indique al jugador el proceso de conexión al host, así como también la cuenta atrás antes de la carrera y la espera de más jugadores si no se ha alcanzado el mínimo número de jugadores necesario. Para ello, creamos dos lienzos nuevos en la escena:

- El lienzo que contendrá los textos asociados a la espera de más jugadores y la cuenta atrás. La clase `LobbyUIManager` se encargará del manejo de está interfaz mediante el seguimiento de los mensajes locales asociados a los eventos `LobbyEvent.WAITING_FOR_MATCH` y `LobbyEvent.MATCH_COUNTDOWN_UPDATE`.

- El lienzo que indicará al jugador en el momento de entrada al *lobby* que se está llevando a cabo la conexión. Dicho lienzo debe ser activado mientras `PlayersFinder` trata de localizar un host, siendo desactivado cuando se encuentre o al finalizar el tiempo de búsqueda e iniciar la conexión como host. Este lienzo será por tanto hijo del objeto PlayersFinder y la clase `PlayersFinder` se encargará de su activación y desactivación.

Las Figuras 24 y 25 muestran un resumen de los objetos resultantes de la transformación de la escena de carrera y el *lobby*. Estos gráficos se centran en mostrar

los cambios hechos con respecto a la versión local, por lo que algunos elementos comunes a ambas versiones y otros menos relevantes han sido omitidos para mejorar su legibilidad.

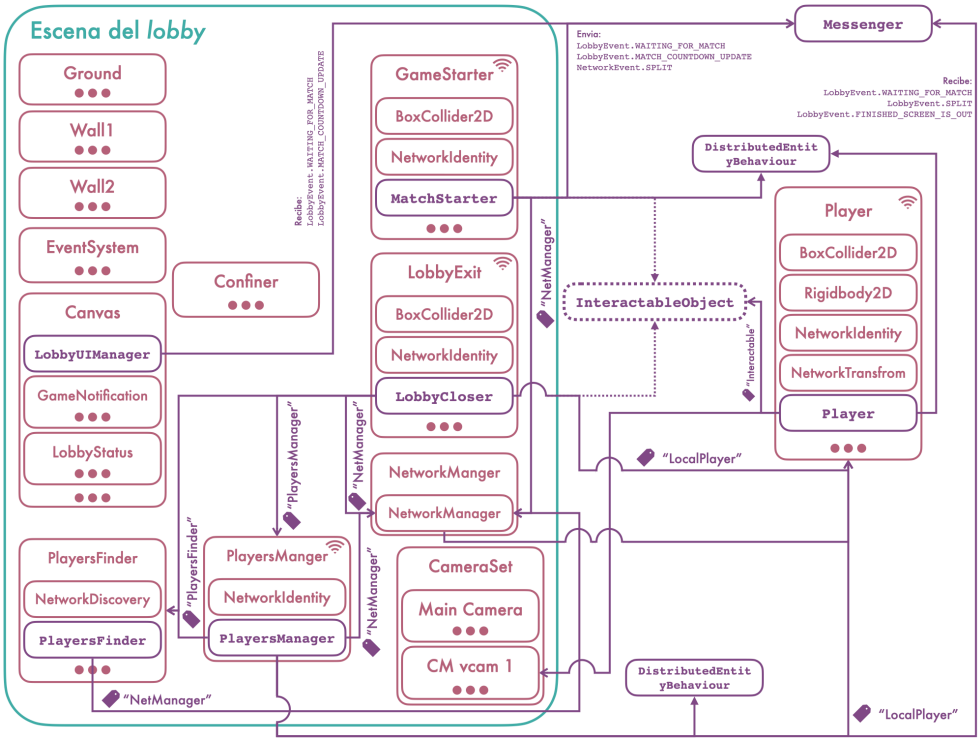

Figura 24: esquema que resume los objetos que conforman el lobby en la versión UNet.

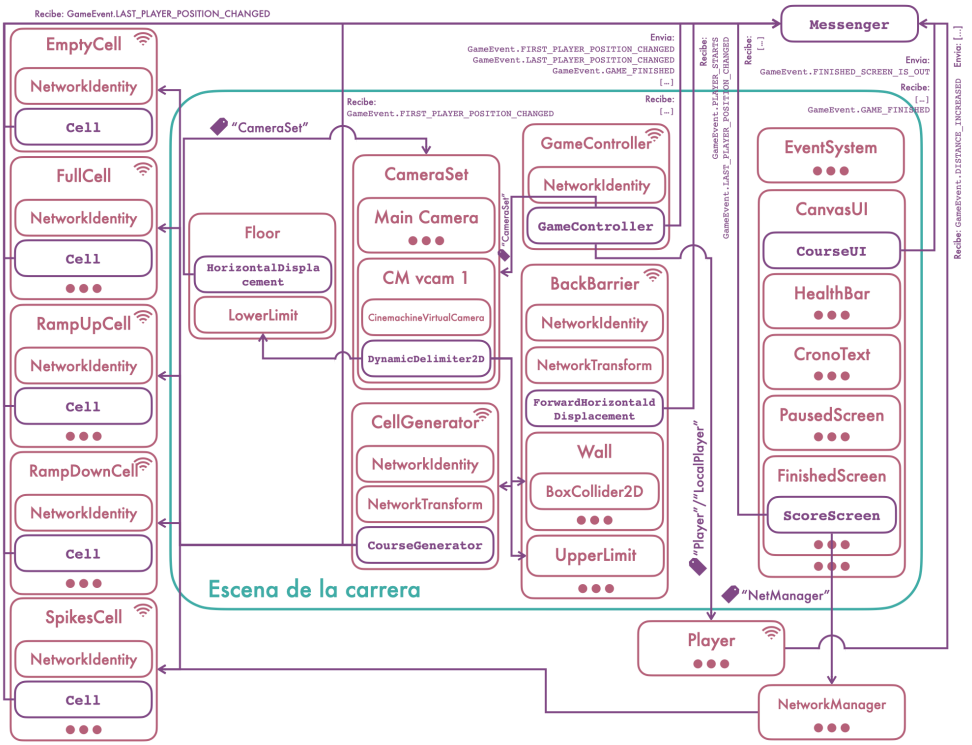

Figura 25: esquema que resume los objetos que conforman la escena de carrera en la versión UNet.

5.3.3.5.Consideraciones adicionales

Antes de cerrar el proceso de adaptación de las diferentes escenas del proyecto a una versión en red, cabe destacar las siguientes consideraciones:

- Para poder indicar en qué posición se debe crear el avatar del jugador de la escena, se pueden emplear objetos que posean un componente NetworkStartPosition. Dicho componente permite que el NetworkManager tome de forma automática su posición como punto de creación para los jugadores. Este sistema permite crear tantos puntos de creación como se desee, pudiendo el NetworkManager escoger el siguiente punto de creación de forma aleatoria o mediante *round robin* según se indique en su campo *Player Spawn Method*.

- Debido a que la posición del objeto BackBarrier depende de la posición del último jugador, cabe la posibilidad de que los jugadores en las primeras posiciones vean un vacío debajo del terreno generado cuando este tenga poca altura. Para solucionar esto, añadimos un nuevo objeto local Floor que se encargará de seguir la posición horizontal de la cámara para situarse siempre debajo del jugador y ocultar el vacío.

- Cuando se produce la separación de los jugadores al iniciar una partida y se nombra un nuevo host para el *lobby*, se debe llevar a cabo la activación de `NetworkDiscovery` en modo host para permitir que futuros clientes puedan conectarse. De igual forma, será necesario que el host de la carrera desactive el envío de mensajes por parte de `NetworkDiscovery` para evitar que nuevos jugadores puedan conectarse directamente a la carrera.

- La escena correspondiente al menú principal es completamente local y todas las conexiones se administran en el *lobby*, con lo que no es necesario realizar ninguna adaptación en el menú principal.

- Durante el transcurso del juego, cabe considerar que pueden darse situaciones de red adversas que provoquen la desconexión de los clientes. Estos eventos pueden ser seguidos gracias al registro de una retrollamada asociada a la recepción de mensajes de tipo `MsgType.Disconnect`, mediante `RegisterHandler(…)` de `NetworkServer` o `NetworkClient`. Cabe destacar que solo puede existir una retrollamada asociada a la recepción de mensajes `MsgType.Disconnect`. Por tanto, cualquier registro sustituirá al anteriormente realizado. Esto implica que al seguir este evento directamente ocultaremos el registro interno de `NetworkManager` para el mismo, causando que los avatares de los clientes no se eliminen tras su desconexión. Para hacer posible la destrucción de estos avatares, será necesario invocar el método `NetworkServer.DestroyPlayersForConnection(msg.conn)` en la retrollamada. La detección de la desconexión nos permitirá poder reiniciar el sistema de descubrimiento en los clientes que pierdan la conexión con el servidor y eliminar los registros asociados al cliente en el host.

5.4.Ampliación de la funcionalidad multijugador

Hasta el momento hemos llevado a cabo la adaptación del juego a su versión multijugador en red mediante la HLAPI de UNet. No obstante, aún no hemos cubierto todos los aspectos de nuestro concepto inicial. Extendemos por tanto la funcionalidad

multijugador de nuestro juego para aumentar la interacción entre los jugadores, incorporando un chat, la posibilidad de definir un nombre de usuario y la capacidad de ver la salud y distancia recorrida de nuestros contrincantes durante la carrera.

5.4.1.Nombre de usuario

Comenzamos por añadir posibilidad de definir un nombre de usuario para el jugador. La definición de un nombre nos permitirá en los siguientes pasos asociar cada mensaje del chat a su emisor correspondiente, facilitando su interpretación, así como también permitiendo diferenciar a los demás jugadores más fácilmente. No obstante, cabe destacar que este nombre de usuario podrá coincidir para diferentes jugadores y cambiarse fácilmente, ya que no se empleará en tareas de identificación o registro, siendo únicamente una ayuda para la identificación de los jugadores por parte del usuario.

El nombre de usuario se definirá en el menú principal del juego permitiendo su modificación rápidamente antes de llevar a cabo la conexión. Incorporamos a la interfaz por tanto un campo para la introducción del nombre de usuario mediante un objeto `InputField`. Creamos también un nuevo objeto para el que definiremos la clase `PlayerNameManager`. De esta clase cabe destacar:

- El campo `nameField`, que contiene la referencia a objeto de la clase `TMP_InputField`. En nuestro caso, pasaremos como valor para este campo el InputField que hemos añadido a la interfaz.
- El método público `SaveName()`, que almacenará el nombre de usuario mediante la llamada a `PlayerPrefs.SetString(…)` siempre que el nombre actualmente almacenado no corresponda con el que se está tratando de guardar.
- En el método `Start()`, se instanciará el texto de `nameField` al actualmente almacenado en las preferencias. Si no hay ningún nombre almacenado, se definirá un nombre aleatorio al juntar la palabra “Player” con un número aleatorio entre cero y 2000 inclusive.

De esta forma, definimos como *callback* para el evento `OnEndEdit()` de `nameField` a la función `SaveName()`. De igual manera, se añadirá también `SaveName()` como *callback* para evento `OnClick()` del botón de jugar. Así, aseguramos que el nombre de usuario se guardará siempre que el jugador inicie una partida o que introduzca un nuevo nombre.

5.4.2.Chat

A continuación, procederemos a la implementación de un chat que permita la comunicación de los usuarios mediante el intercambio de mensajes de texto. Para ello, es necesaria la creación de dos nuevas clases:

- Por una parte, necesitamos administrar la representación en pantalla de los mensajes recibidos a través del chat. Para ello, creamos la clase `ChatDisplay`. Esta clase se encargará de mostrar los mensajes en la interfaz de jugador, por lo que será local a cada jugador y heredará de `MonoBehaviour`. Para poder mostrar los mensajes, la clase contendrá referencias a los elementos necesarios de la interfaz como la caja de contenido que se encargará de contener los mensajes de texto o el *prefab* empleado para representar los mensajes. De esta manera, define el método público

`AddMessage(string msg, uint id)` que se encargará de instanciar un mensaje nuevo, modificar su contenido con el mensaje proporcionado y vincular el mensaje como último hijo de la caja de contenido para que aparezca al final de la lista de mensajes. El método también toma como argumento un identificador que indicará el `netId` del avatar del jugador que lo envió, permitiendo así facilitar el seguimiento del chat, ya que los mensajes escritos por el jugador local se alinearán a la izquierda en lugar de a la derecha. Podremos ocultar el chat al pulsar la tecla M.

- Por otra parte, necesitamos una clase que se encargue de la sincronización e intercambio de los mensajes del chat entre los diferentes jugadores. Por tanto, esta clase deberá tener un comportamiento en red y permitir a las instancias cliente llevar a cabo operaciones sobre el servidor para transmitir los mensajes escritos por los jugadores. De esta forma, creamos la clase `ChatManager` que heredará de `DistributedEntityBehaviour`. Esta clase contendrá una referencia a un objeto `TMP_InputField`, de tal forma que cuando el jugador termine la edición de los contenidos del campo, la clase `ChatManager` se encargará de la obtención y sincronización del texto. De igual forma, también contendrá una referencia a una instancia de `ChatDisplay` para solicitar la representación de los mensajes recibidos. Si el jugador host escribe un mensaje, se invocará la RPC `RpcSendMessage(string msg, uint id)` de forma que todos los clientes reciban el mensaje, la cual ejecutará a su vez el método `AddMessage(…)` de `ChatDisplay`. No obstante, cuando sea un jugador cliente el que escriba un mensaje, será necesaria la llamada del comando `CmdSendMessage(string msg, uint id)` para poder sincronizar el mensaje. Esto se debe a que no existe sincronización directa entre las instancias cliente, por lo que es preciso solicitar al servidor que lleve a cabo la sincronización. Por tanto, este comando se encargará a su vez de llevar a cabo la RPC en la instancia servidor.

Gracias a estas clases y a un nuevo lienzo que incorpora todos los elementos necesarios que formarán el chat, ya será posible la comunicación mediante mensajes de texto entre los jugadores. Una captura del chat resultante puede verse en la Figura 26. Para permitir que los jugadores puedan seguir interactuando también durante la partida, se asigna al chat la propiedad `DontDestroyOnLoad`, permitiendo que sobreviva a la transición entre escenas y siendo necesario eliminar el objeto antes de la vuelta al *lobby* para evitar duplicados.

Un inconveniente que surge de la incorporación de un chat de texto al juego es el control accidental del avatar al escribir en él. El componente InputField de Unity permite la escritura de texto, pero no se encarga de evitar que la entrada de texto llegue a otros componentes mientras se escribe [35], con lo que al escribir en el chat se pueden producir interacciones o acciones indeseadas.

La solución a este problema pasa por el diseño de una clase local llamada `InputAvailabilityManager`, que se utilizará para poder consultar en cualquier momento la validez de la entrada. Cuando el usuario esté escribiendo en el chat, la entrada no será válida para el resto de componentes, y cuando el usuario termine de escribir, será válida. La clase `InputAvailabilityManager` contará con los métodos públicos `UserStartedTyping(…)` y `UserFinishedTyping(…)` que permitirán actualizar el estado de la entrada. Mediante la propiedad pública de lectura `UserIsTyping` será posible consultar el estado. Así, gracias a la incorporación de esta

clase como hijo del lienzo del chat, podemos asignar sus métodos como *callbacks* de los eventos *On Value Changed* y *On End Edit* respectivamente del `InpuField` del chat.

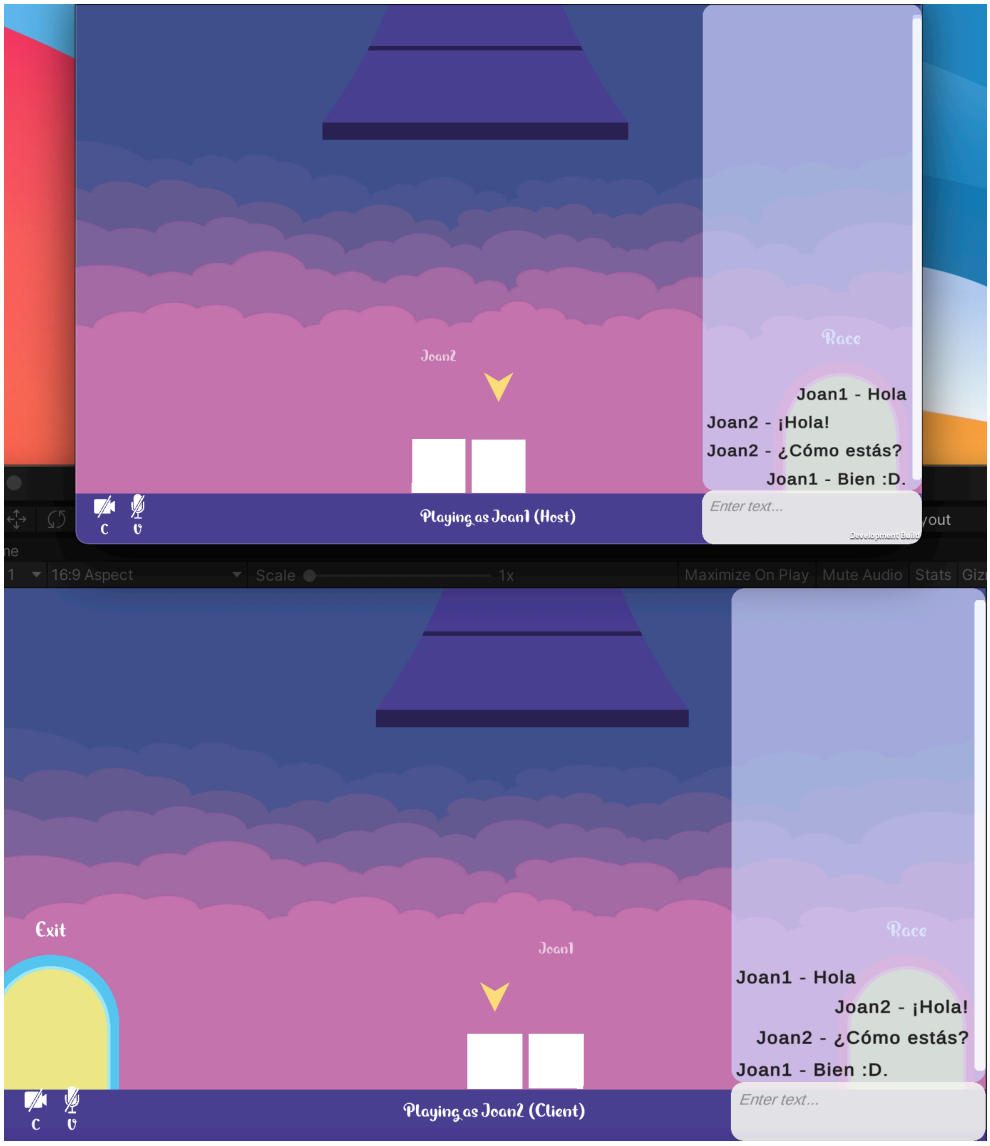

Figura 26: uso del chat para la interacción entre dos usuarios.

Mediante el uso de la etiqueta “InputAvailabilityManager” para este objeto, cualquier elemento que necesite hacer uso de la entrada del teclado, como por ejemplo la clase `Player`, deberá comprobar si existe en la escena un objeto `InputAvailabilityManager` y consultarle en caso afirmativo el estado de la entrada antes de procesarla.

5.4.3. Información del estado de los jugadores

Con el objetivo de incrementar la competitividad entre los jugadores y el interés en la carrera, incorporamos sobre los jugadores una pequeña interfaz encargada de mostrar la salud del jugador, su distancia recorrida y su nombre de usuario. Para ello, procederemos a llevar a cabo las siguientes modificaciones sobre el *prefab* del avatar del jugador:

- Añadimos como hijo del objeto Player un objeto PlayerTags situado en el centro del mismo que se encargará de contener la información relativa al jugador. El uso de este objeto para englobar la información es importante, ya que para su correcta lectura será necesario que la información no gire con la rotación del jugador. Para ello, PlayerTags se encargará de contrarrestar la rotación del jugador gracias al *script* local *PlayerTag.cs*.

- Incorporamos como hijos de PlayerTags los lienzos necesarios para la representación de una pequeña barra de salud, el texto indicando la distancia recorrida y el texto relativo al nombre de usuario. Para facilitar la identificación del avatar correspondiente al jugador local, añadimos también un flecha que apuntará hacia el avatar local.

- Modificamos la clase `Player` para que lleve a cabo la representación de la información. Para ello:

    - Cuando la instancia pertenezca al jugador local, se ocultará la información relativa a la salud, la distancia recorrida y el nombre de usuario, mostrándose únicamente la flecha que indica que se trata del jugador local. Esto se debe a que la información al respecto de la salud y la distancia del jugador local ya se muestra en la interfaz de la carrera.

    - Cuando la instancia pertenezca a un jugador no local se ocultará la flecha. La salud y la distancia también se ocultarán cuando la escena activa sea el *lobby*. Para actualizar de los indicadores de salud y distancia recorrida, la clase `Player` llevará a cabo la ejecución del comando `CmdUpdateHealth(int health)` que se encargará a su vez de la ejecución de la RPC `RpcUpdateHealthBar(int h)` que permitirá la actualización de la barra de salud cuando no se ejecute en la instancia del jugador local. De igual forma, el seguimiento de los mensajes locales asociados a `GameEvent.DISTANCE_INCREASED` permitirá la invocación del comando `CmdDistanceIncreased(int d)` que a su vez ejecutará la RPC `RpcDistanceIncreased(int d)` encargada de sincronizar el texto de la distancia recorrida en la instancias no pertenecientes al jugador local.

5.4.4. Tabla de resultados

Una característica común en los juegos multijugador es la presencia de algún estilo de tabla de clasificación para poder ver las puntuaciones de los demás jugadores, ya sea a nivel global de todo el juego, o a nivel local de la partida desarrollada. Nuestro juego hasta el momento únicamente muestra al jugador su distancia recorrida al terminar la partida, pero esta funcionalidad se puede extender para que se proporcione una lista con las distancias recorridas por todos los jugadores en la partida. Optamos por tanto únicamente por una tabla de resultados local a la partida realizada, debido a que nuestro juego no emplea ningún tipo de servidor dedicado donde almacenar las puntuaciones de todos los jugadores.

Para poder implementar una tabla de resultados, es un requisito fundamental que todos los jugadores dispongan de la puntuación obtenida por los demás. Tenemos por tanto diferentes alternativas:

- Una opción es registrar de forma local en cada jugador la distancia recorrida por las instancias de los demás jugadores y emplear estos registros para formar la tabla de resultados. Esta opción tiene la ventaja de no necesitar un intercambio adicional de información y mensajes entre los clientes al que ya tenemos. No obstante, la sincronización de la posición de un jugador en red puede sufrir de imprecisiones y retrasos, lo que podría provocar una discrepancia de los resultados para el mismo jugador en los diferentes clientes.

- Otra alternativa es llevar a cabo el registro de todas las distancias recorridas por los avatares en el host, sincronizando dichas distancias a todos los clientes. Esta opción es similar a la anterior, pero no se producen divergencias de resultados, aunque sí que siguen pudiendo haber retrasos e imprecisiones con la medida llevada a cabo por el host.

- Finalmente, tenemos la posibilidad de aprovechar el cálculo de la distancia recorrida por el jugador local que se lleva a cabo en cada máquina y, una vez obtenido, transmitirlo al resto de jugadores. Esta alternativa implica un coste adicional de mensajes y sincronización, pero garantiza la precisión de los resultados obtenidos en todos los jugadores.

Por lo tanto, decidimos optar por la última opción. Una dificultad que se presenta en el desarrollo de esta alternativa es que la distancia recorrida no se calcula hasta que el jugador muere para asegurar que se obtiene la medida de forma precisa. Esto lleva a tener que proceder a la sincronización de la distancia recorrida por un jugador tras la muerte del mismo, lo que es imposible según el modelo actual. Cabe recordar que es necesario, para poder proceder a la modificación de una instancia servidor por parte del cliente, que el objeto jugador nos conceda la autoridad necesaria. Pero, tras llegar su salud a cero, el jugador es destruido por lo que no será posible obtener la autoridad necesaria y la distancia recorrida no se podrá comunicar al host para su difusión.

La forma correcta de proceder deberá ser, por tanto, ocultar el avatar del jugador en todas las instancias mientras esperamos a que la puntuación obtenida se sincronice. Una vez sincronizada la puntuación, podremos proceder a la destrucción del jugador. En consecuencia, llevaremos a cabo los siguientes pasos:

- En el comando `CmdUpdateHealth(…)` de la clase `Player`, en lugar de llamar a `NetworkServer.Destroy(gameObject)` procederemos a la llamada de `RpcHidePlayer()`. Esta RPC se encargará de ocultar el jugador en todos los clientes.

- Crearemos la clase `ScoresManager`, la cual se encargará de llevar a cabo la sincronización de la distancia obtenida por cada jugador y de almacenar las distancias recorridas por todos los jugadores. Esta clase será una clase en red que llevará a cabo sincronizaciones por lo que heredará de la clase `DistributedEntitieBehaviour`. Sus aspectos más importantes son:

    - La lista `scores` que almacena tuplas de la forma (cadena, entero) que asocian el nombre de un jugador y la distancia que ha recorrido. Para poder asegurar que la lista estará ordenada, hacemos uso de la extensión de la clase `List` de Jackson Dunstan [19]. Esta extensión incorpora el método `InsertIntoSortedList(…)` que nos permite insertar las tuplas de forma ordenada en la lista según su componente entera.

- Los campos `numOfScoresNeeded` y `currentScoresReceived`, que determinan el número de puntuaciones que debemos esperar y el número actual de puntuaciones registradas. El número de puntuaciones necesarias será igual al número de jugadores, con lo que se podrá obtener a partir de la propiedad de lectura pública `NumberOfPlayers` de `PlayersManager`.

- Una referencia al `ScoreScreen` que se encargará de mostrar la tabla de resultados.

- Los métodos `CmdSyncScores(string player, int d)` y `RpcSyncScores(string player, int d)`, los cuales se encargarán de la sincronización de la distancia recorrida por cada jugador entre todas las instancias y la destrucción del avatar del jugador cuya puntuación se ha obtenido.

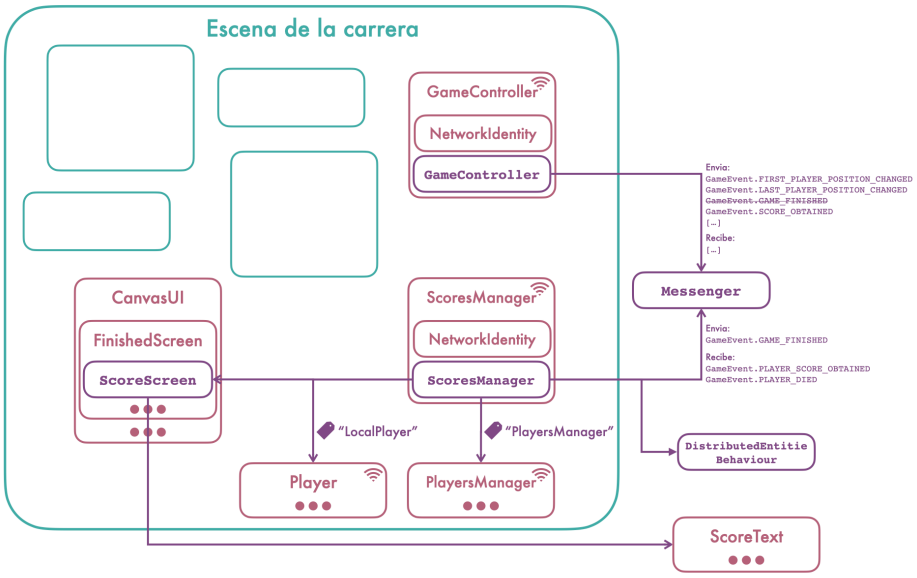

Figura 27: esquema resumen de la incorporación de ScoresManager en la versión UNet.

De esta manera, las instancias de la clase `ScoreScreen` se suscribirán a los mensajes locales asociados al evento `GameEvent.PLAYER_SCORE_OBTAINED`, que serán emitidos por `GameController`. La recepción de este evento, el cual transmitirá la tupla (nombre, distancia), causará la sincronización del resultado mediante la llamada al comando `CmdSyncScores(…)`. Cuando se reciban todas las puntuaciones esperadas, se proporcionarán los resultados a la instancia de `ScoreScreen` disponible y se emitirá el mensaje local asociado a `GameEvent.GAME_FINISHED`. Cabe destacar que la emisión de este mensaje sustituye a la emisión del mismo en la clase `GameController` para asegurar que la desconexión se llevará a cabo tras haber recibido las puntuaciones de todos los jugadores.

- Finalmente, añadiremos los elementos de interfaz necesarios en la escena de juego para mostrar las puntuaciones de todos los jugadores. La construcción de esta tabla de puntuaciones se asemeja a la construcción del chat, de forma que el método

público `AddScore(string player, int d)` permite la incorporación de una nueva fila a la tabla y será el llamado por `ScoresManager`.

Gracias a estos cambios, resumidos en la Figura 27, ya es posible la visualización de una tabla de resultados al terminar la partida. Cabe destacar que será necesario incluir a la escena de juego un objeto que incorpore el *script ScoresManager.cs*, completando las referencias necesarias. Dicho objeto deberá poseer un componente NetworkIdentity con la propiedad *Local Player Authority* activa para que sea posible la sincronización de las puntuaciones.

5.4.5.Comunicación audiovisual

Uno de los propósitos de un juego multijugador es conectar a las personas y permitir la interacción con familia y amigos en un entorno de diversión. Con esta idea en mente y reforzada por la actual situación global, nos planteamos la incorporación de funcionalidades de transmisión tanto de vídeo como de audio para explorar las capacidades de UNet en este ámbito y permitir a los jugadores una mayor capacidad de comunicación.

5.4.5.1.Implementación base para la comunicación

Primero, plantearemos el modelo que queremos conseguir. Un jugador podrá activar la webcam y/o el micrófono de su portátil y transmitir la imagen y/o audio de los mismos a los demás jugadores. Para ello, será necesario de forma periódica capturar la imagen o el sonido, transmitir los mismos a la instancia servidor (paso que se omite si la captura se toma directamente en esta instancia) y, finalmente, transmitir la imagen o el sonido del host a todos los clientes para su visualización o reproducción. Cabe destacar que dicha operación deberá hacerse por cada muestra que se quiera transmitir, por lo que deberemos ajustar el número de muestras por segundo a transmitir para no inundar la red.

Debido a que la transmisión de la imagen y el audio poseerán muchas partes en común, comenzaremos analizando las necesidades comunes de ambos e implementándolas en una clase común. Una primera opción que surge para poder llevar a cabo la transmisión de audio o imagen es emplear el mecanismo de comandos y RPC que hemos empleado hasta el momento. No obstante, dicha opción no es apropiada ya que este tipo de funciones están pensadas para el envío de argumentos de funciones, por lo que hacen uso de un tamaño de paquete menor. Por tanto, la alternativa apropiada es la transmisión directa de la imagen o el sonido mediante mensajes. Mediante la herencia de la clase abstracta `MessageBase`, podemos crear nuevas clases de mensajes a transmitir mediante la red.

Así, será preciso realizar la transmisión directa de mensajes que contengan los datos necesarios para la reconstrucción del medio correspondiente en cada cliente. Cabe destacar como referencia para el diseño de este modelo de mensajes el foro de Unity acerca de la transmisión de vídeo a través de la red, en especial las contribuciones del usuario Donnysobonny [37]. Sus aportaciones plantean los retos y los efectos adversos que se pueden presentar en la transmisión de una secuencia de imágenes a través de la red, describiendo genérica y verbalmente el modelo necesario para que el sistema esté preparado para estas situaciones. De esta manera, gracias a este foro identificamos las siguientes adversidades:

- Las imágenes o audios podrían ser demasiado grandes para su transmisión en un solo mensaje. Será por tanto necesario en estos casos llevar a cabo la fragmentación del mensaje. Para ello, se propone dividir la transmisión de la información en un mensaje inicial de cabecera, que contendrá la información necesaria para preparar la recepción de los datos, y a continuación, los mensajes correspondientes a los fragmentos de datos que forman el medio correspondiente. Para poder vincular cada cabecera con sus fragmentos asociados, todos los mensajes correspondientes a la misma muestra deberán poseer un identificador propio. Además, se indica que cada fragmento deberá contener también un entero que indique su orden relativo y permita la correcta reconstrucción de la secuencia de fragmentos. [37]. No obstante, además de estas ideas, consideramos necesario incluir también el tamaño máximo de los fragmentos para facilitar el proceso de reconstrucción y el `netId` correspondiente con el avatar del jugador que lleva a cabo la transmisión como elemento de seguridad para verificar su origen.

- Los mensajes podrían llegar de forma desordenada. Esto implica que los jugadores podrían recibir las muestras en el orden equivocado. Para solventarlo, se propone asociar un valor que indique la ordenación de cada muestra [37]. Nosotros optamos por incluir en los mensajes cabecera una marca de tiempo que permita la ordenación correcta de los mensajes recibidos para cubrir esta necesidad. Por otro lado, también nos dimos cuenta de que esto implica que es posible que los mensajes fragmento lleguen antes que sus correspondientes cabeceras, por lo que vimos necesario almacenar todos los fragmentos cuyo identificador sea desconocido hasta que se reciba su cabecera correspondiente.

- Finalmente, nosotros determinamos también la posibilidad de que los mensajes no lleguen o que tarden mucho tiempo en llegar. Por este motivo, será preciso aplicar una cuenta atrás a cada cabecera recibida, de forma que si no se han recibido todos los fragmentos correspondientes a la muestra tras la cuenta atrás, esta muestra se descarte.

Así, partiendo de este foro de Unity [37] como base y completando el modelo planteado, definimos el sistema de mensajes que nos permitirá la transmisión tanto de la voz como de la imagen. Un último componente que se propone en [37] es un tercer tipo de mensajes para indicar el fin de la secuencia de fragmentos enviados. No obstante, en nuestro caso concluimos que este no era necesario, ya que gracias a la información incluida en la cabecera podemos determinar el tamaño de la muestra esperada y cuándo la hemos recibido completamente.

Una vez establecido el sistema de mensajes a utilizar, hay que determinar el papel del emisor y los receptores de los mismos:

- El jugador que transmite las muestras capturadas por su cámara o micrófono deberá realizar de forma periódica el envío de la información capturada. Para no saturar la red, será necesario especificar el número de transmisiones por segundo a llevar a cabo. En cada transmisión, la instancia local se encargará de descomponer el medio capturado en un paquete cabecera y tantos paquetes fragmento como sea necesario. Para cada muestra a enviar, se deberá crear un identificador propio, determinar su marca temporal y el tamaño de los fragmentos a enviar. Todos los mensajes se deberán transmitir al host.

- El host recibirá la transmisión del jugador, y su papel será reenviar todos los mensajes que reciba por parte del jugador al resto de jugadores.

- Los jugadores esperarán la recepción de los mensajes de la transmisión. De esta manera, al recibir un mensaje cabecera, se registrará una nueva muestra, asociando el identificador proporcionado con sus datos correspondientes. Por cada cabecera recibida, deberá comprobarse tanto si se dispone de fragmentos con el mismo identificador que hayan llegado de forma prematura como su marca temporal para determinar su posición en la secuencia de muestras. Todos los fragmentos cuya marca temporal sea anterior a la muestra que actualmente se quiere mostrar/reproducir será descartado. Se creará entonces una nueva corrutina, la cual se encargará de esperar la recepción de los fragmentos asociados con dicho identificador. Si tras un tiempo de espera determinado no se han recibido todos los fragmentos, se descarta la información relativa a ese medio. Cuando se recibe un fragmento, se comprueba si su identificador ha sido ya registrado. En caso afirmativo, los datos del fragmento se almacenan para reconstruir progresivamente el medio, de forma que cuando se reciban todos los datos, el medio se marcará como listo para regenerar. En caso contrario, se almacenará temporalmente el fragmento. Finalmente, todos los medios completamente recibidos se reconstruirán según el tipo deseado para utilizarlos.

Una vez definido el modelo de sincronización que vamos a utilizar, podemos proceder a la implementación de las partes comunes tanto a la transmisión de imagen como de sonido en una clase padre. Primero determinamos las características del objeto que incorporará el *script* encargado de la transmisión. Este objeto deberá tener un comportamiento en red y cada jugador deberá tener una copia propia del mismo, ya que cada copia administrará la transmisión de un jugador. Por tanto, será el propio avatar del jugador el que deberá implementar el *script* de retransmisión.

Así, elaboramos la clase abstracta `StreamManager` que hereda de `NetworkBehaviour` y define internamente todos los rasgos comunes y necesarios para la transmisión de cualquier medio. Las definiciones necesarias son:

- La clase abstracta protegida `StreamMsgType` que contendrá los campos `Header` y `Chunk` de tipo short. Estos campos se emplearán para identificar los mensajes cabecera y fragmento respectivamente. Ambos se utilizarán para registrar las funciones retorno asociadas a cada tipo de mensaje, por lo que es necesario que sus valores sean diferentes. Además, no solo deben ser diferentes entre sí, sino que también deberán ser diferentes entre jugadores, de tal manera que cada par de identificadores cabecera y fragmento debe ser único. Para ello, definimos el método abstracto `UpdateTypes(int n)` que dado el `netId` del avatar del jugador asignará un identificador propio a los campos `Header` y `Chunk`. Esta clase está pensada para ser implementada en cada tipo de retransmisión definiendo sus propios valores iniciales de `Header` y `Chunk` y su propia actualización de valores según el `netId`, de forma que se garantiza que los identificadores de cada tipo de transmisión serán diferentes entre sí también.

- La clase protegida `StreamHeaderMessage` que hereda de `MessageBase`. Esta clase contiene los campos públicos:

- `netId`, que será el identificador del avatar del jugador que transmite esta cabecera.
- `id`, que será el identificador propio de la imagen o el audio que representa.
- `chunkSize` para almacenar el tamaño máximo que tendrán los fragmentos asociados a la cabecera.
- `timeStamp`, que contendrá la marca de tiempo en la que se crea la cabecera.

Así, definiremos un constructor vacío para esta clase que no tomará argumentos, ya que es preciso para poder usarla como mensaje a transmitir, y un constructor que tomará los argumentos necesarios para inicializar los campos. El campo `timeStamp` se inicializa en dicho constructor con el valor de `System.DateTime.Now.ToUniversalTime().Ticks` que nos permite obtener la fecha y hora actual. La clase `StreamHeaderMessage` también implementará los métodos `Serialize(…)` y `Deserialize(…)` [47], llevando a cabo la escritura y lectura de los campos respectivamente gracias a las funciones de `NetworkWriter` y `NetworkReader`. Hay que tener en cuenta que la lectura debe hacerse en el mismo orden que la escritura para asegurar que se inicializan los campos de forma correcta. La clase `StreamHeaderMessage` está pensada para ser heredada y poderse emplear para transmitir la información de cabecera adicional específica de cada medio.

- La clase protegida `StreamChunkMessage` que hereda de `MessageBase`. Esta clase también contiene los campos públicos `netId` e `id`, pero además:
  - `order`, que indicará su posición correspondiente en la secuencia de fragmentos, empezando por el cero.
  - `size`, que indicará el tamaño real de la información contenida en el mensaje.

  Esta clase definirá, de igual manera que `StreamHeaderMessage`, un constructor vacío sin argumentos y los constructores necesarios para inicializar los campos. Esta clase también implementa de igual modo los métodos `Serialize(…)` y `Deserialize(…)` de `MessageBase`. Estos métodos no requieren ser implementados siempre y se puede confiar en la generación automática para los mismos que proporciona UNet [47]. No obstante, al tratarse de clases pensadas para ser heredadas, hemos considerado más apropiada su implementación para tener un mayor control y seguridad sobre la misma.

Un aspecto muy importante a definir también en la clase `StreamManager` son las estructuras de datos que nos permitirán el almacenamiento y administración de la información recibida a través de la retransmisión. Dichas estructuras serán las mismas independientemente del tipo de transmisión, pero los tipos de datos que contendrán cambiarán. Por ello, decidimos implementar una clase con tipos genéricos para contener y administrar dichas estructuras de datos llamada `StreamMessageDataSupport`. El motivo de la implementación de las estructuras de datos dentro de una clase privada y no directamente dentro de `StreamManager` es la necesidad de emplear tipos genéricos para dichas estructuras y la imposibilidad de diseñar clases genéricas que hereden de `NetworkBehaviour` por la generación automática de código en la que confía UNet.

La clase `StreamMessageDataSupport` define los tipos genéricos `struc`, para representar el tipo asociado a la estructura que se empleará para reensamblar el medio transmitido, `headerMsg`, para representar la clase de mensajes cabecera utilizados, y `chunkMsg`, para representar la clase de mensajes fragmento utilizados. Los tipos `headerMsg` y `chunkMsg` deberán ser derivados de `StreamHeaderMessage` y `StreamChunkMessage` respectivamente. Así, las estructuras de datos que contendrá `StreamMessageDataSupport` son:

- El diccionario `streamWasFullyReceived`, que contendrá elementos de tipo booleano indexados mediante un entero sin signo. De esta manera, podemos vincular el identificador de una muestra con su estado y consultar si ha sido recibida completamente.

- El diccionario `streamData`, que contendrá las muestras recibidas en formato `struc` indexadas mediante su identificador. Esta colección nos permitirá almacenar temporalmente las muestras recibidas mientras se completan con sus fragmentos.

- La lista `unheadedChunks`, que contendrá tuplas del tipo `(struc, float)`. Esta lista permitirá almacenar los mensajes recibidos de forma prematura ya que su identificador no ha sido registrado todavía. La tupla se emplea para poder asociar a cada mensaje el tiempo que ha pasado desde que se recibió, de tal manera que si el tiempo supera un límite establecido, este fragmento se descartará. El uso de una lista en lugar de un diccionario se debe a la necesidad de poder almacenar varios fragmentos que puedan llegar asociados al mismo identificador.

- El diccionario `amountOfEarlyChunks` que contendrá el número de fragmentos prematuros recibidos asociados a su identificador. Esto nos permite consultar fácilmente si hay fragmentos prematuros cuando se recibe una cabecera, sabiendo además cuántos fragmentos debemos recuperar.

- La colección `streamIdsReceived`. Esta colección se encarga de almacenar los identificadores recibidos y que no han sido todavía descartados. Estos identificadores deben ordenarse según la marca de tiempo indicada en la cabecera para garantizar el correcto orden de reproducción de las muestras. Para poder conseguir una lista ordenada según un valor adicional, diseñamos la clase `KeySortedList<T, D>`, la cual tendrá una lista de llaves y una lista de valores, de tal forma que las llaves se insertarán de forma ordenada mediante `InsertIntoSortedList(…)` y los valores se insertarán en la misma posición que las llaves. Gracias a esta clase, podemos almacenar los identificadores ordenados según su marca temporal.

- El número en coma flotante `unheadedChunksTimeout`. Este campo se utiliza como parámetro para ajustar el tiempo máximo que un fragmento sin cabecera puede ser almacenado.

Todas estas estructuras se definen como campos privados en la clase. Por tanto, definimos en esta clase todo el conjunto de accesores y métodos necesarios para acceder y modificar la información almacenada. Cabe destacar los siguientes:

- La propiedad `this[uint id]` que permite obtener o modificar el objeto `struc` asociado a un identificador .

- La propiedad `FirstId` que nos permite obtener la primera muestra en la secuencia recibida.
- La propiedad `FirstTimestamp` que nos permite obtener la marca temporal de la primera muestra en la secuencia recibida.
- El método `RemoveStream(uint id)` que nos permite eliminar los datos asociados a un determinado identificador.
- El método `RecoverEarlyChunks(headerMsg header, struc streamS, StreamChunkHandler<chunkMsg> SaveChunk)` que dado un mensaje cabecera, la estructura generada a partir de él para contener los datos y una función que dado un mensaje fragmento actualiza dicha estructura, se encarga de registrar la estructura y buscar posibles fragmentos prematuros pertenecientes a esta cabecera para la recuperación de sus datos.
- El método `CheckTimestamp(headerMsg header)` que comprobará que la marca temporal de la cabecera proporcionada es mayor que la de la primera muestra en la secuencia actual antes de registrar el identificador devolviendo verdadero. En caso contrario, se descarta y se elimina cualquier información vinculada a su identificador devolviendo falso.
- El método `AddChunk(chunkMsg chunk, StreamChunkHandler<chunkMsg> SaveChunk)` que se encarga de actualizar la muestra asociada a partir de los datos contenidos en un mensaje fragmento haciendo uso de la función proporcionada `SaveChunk`.
- El método `ManageUnheadedChunks()` que se encargará de controlar el tiempo que han permanecido almacenados los fragmentos prematuros y de eliminarlos cuando se supere el tiempo máximo establecido. Esta función está pensada para ser llamada en cada ejecución de un método `Update()`.
- La corrutina `WaitTillReceiveAllTheStream(headerMsg header, WaitingCondition Condition, float maxWaitingTime)` que dado un mensaje cabecera, una condición de espera y un tiempo máximo de espera, se encarga de esperar la recepción de los fragmentos asociados a la cabecera hasta que la condición sea falsa o que se supere el máximo tiempo establecido. Si se reciben todos los fragmentos a tiempo, se marcará la muestra como lista para regenerar, y se descartará en caso contrario.

La clase `StreamManager` también define además una serie de campos y métodos para abstraer las tareas comunes necesarias para la transmisión de un medio independientemente del tipo de medio que sea. De entre estos, cabe destacar los siguientes:

- El campo protegido `maxChunkSize` empleado para definir el tamaño máximo de los fragmentos a transmitir.
- Los campos protegidos `streamTimeout` y `pendingHeadersTimeout`, que permiten ajustar el tiempo máximo de espera a los fragmentos de una muestra y el tiempo máximo que se puede conservar un fragmento prematuro.

- El campo protegido `transmissionsPerSecond` para definir el número de transmisiones a realizar por segundo.

- El campo protegido `isStreamOn`, que se modifica mediante el atributo `[SyncVar]` para ser empleado como mecanismo de sincronización entre las instancias en red del objeto, permitiendo a todas las instancias conocer cuando la transmisión está en marcha.

- Para permitir la recepción de los mensajes asociados a un tipo concreto, es necesario llevar a cabo el registro de las funciones encargadas de recibir dicho tipo de mensajes. El método `CreateHandlers(…)` permite realizar este registro al recibir una clase que hereda de `StreamMsgType` y las funciones responsables de administrar la recepción de mensajes cabecera en el servidor, mensajes fragmento en el servidor, mensajes cabecera en el cliente y mensajes fragmento en el cliente. Cuando dicho método se ejecute en la instancia servidor, el registro se lleva a cabo mediante `NetworkServer.RegisterHandler(…)`, mientras que para las instancias clientes se lleva a cabo a través de `NetworkManager.singleton.client.RegisterHandler(…)`. Como `NetworkServer` es común a todos los jugadores y `NetworkManager.singleton.client` es compartido por todas las instancias de objetos en el mismo cliente, es importante que cada jugador tenga un valor propio y único para sus tipos cabecera y fragmento, ya que si coinciden se sobrescribe el registro.

- Como operación contraria al registro de las funciones encargadas de recibir mensajes, tenemos el método `DestroyHandlers(…)` que tomando como argumento el tipo `StreamMsgType` de los mensajes empleados se encarga de dar de baja la función asociada a ese tipo. Esto se consigue gracias a la función `UnregisterHandler(…)` de `NetworkServer` o `NetworkManager.singelton.client` según corresponda.

- Para la transmisión de las muestras capturadas tenemos el método `BroadcastStream()`, que al ser llamado una vez en cada ejecución de un `Update()` permite el envío según el número de transmisiones por segundo que se hayan establecido.

- La corrutina abstracta `SendStream()` deberá ser implementada en las clases que hereden de `StreamManager`. Esta corrutina debe encargarse de capturar la muestra que queremos enviar y procesarla para poder enviar los respectivos mensajes cabecera y fragmento con la información correspondiente. El método `BroadcastStream()` se encarga de la ejecución de `SendStream()`.

- La actualización de la muestra a regenerar actualmente se lleva a cabo en el método `UpdateStream(…)` que dada la `StreamMessageDataSupport` que contiene la información recibida y una función capaz de regenerar la muestra en el objeto deseado dado su identificador, se encarga de llevar a cabo la regeneración de la primera muestra en la secuencia recibida cuando está completa, siempre que la transmisión esté en marcha.

- Para el envío de mensajes cabecera y mensajes fragmento incluimos los métodos `SendHeaderMessage(…)` y `SendChunkMessage(…)`, los cuales toman como argumentos el tipo `StreamMsgType` de los mensajes empleados y el mensaje a

enviar. Estos métodos se encargan de llamar a las instrucciones apropiadas según se ejecuten en la instancia servidor para transmitir a todos los clientes o se ejecuten en la instancia cliente para la transmisión al servidor. Para la primera, empleamos internamente el método `NetworkServer.SendByChannelToReady(…)` que permite la propagación de los mensajes a todos los clientes cuya instancia esté lista a través del canal especificado. Para la segunda, empleamos `NetworkManager.singleton.client.SendByChannel(…)` que permite el envío del mensaje proporcionado a través del canal indicado.

- Para simplificar la tarea de propagación de los mensajes recibidos por el servidor, creamos los métodos `OnStreamHeaderMessageFromClient(…)` y `OnStreamChunkMessageFromClient(…)` que toman como argumentos el tipo de mensajes empleados y el respectivo mensaje a propagar. Estos métodos se han diseñado con la idea de ser llamados en la ejecución de las funciones que administran la recepción de los mensajes por parte del servidor. El motivo por el cual no se registran directamente estas funciones como receptores de los mensajes se debe al tipo de argumentos necesarios. Los administradores deben tomar únicamente un argumento de tipo `NetworkMessage`, mientras que para la propagación de los mensajes es necesario un mensaje `StreamHeaderMessage` o `StreamChunkMessage` además del tipo `StreamMsgType` de mensajes empleados.

- Con la misma idea en mente, creamos `OnStreamHeaderMessageFromServer(…)` y `OnStreamChunkMessageFromServer(…)` para administrar la recepción de mensajes en las instancias clientes. Estos métodos toman como argumentos el mensaje correspondiente y una función a la que se llamará para proceder al procesamiento de los mismos. La necesidad de tomar esta función como argumento se debe a que el tipo de procesamiento que se lleva a cabo dependerá del medio a transmitir. Tanto estos métodos como `OnStreamHeaderMessageFromClient(…)` y `OnStreamChunkMessageFromClient(…)` se encargarán de comprobar que el `netId` proporcionado en el mensaje coincida con el `netId` actual antes de proceder a procesar o propagar los mensajes.

Para terminar con la implementación de la clase `StreamManager`, diseñamos la interfaz `IMediaInputManager` que define los métodos `StartRecording()` y `StopRecording()`. Esta interfaz debe ser implementada por todas aquellas clases que permitan la transmisión de un medio y los métodos `StartRecording()` y `StopRecording()` permitirán el control de dicha transmisión. Por tanto, `StreamManager` incorporará esta interfaz definiendo los métodos como abstractos, ya que su implementación dependerá del tipo de medio a transmitir.

5.4.5.2.Streaming de imagen

El siguiente paso es llevar a cabo la implementación de las clases responsables de la transmisión de la cámara y el micrófono partiendo de la base proporcionada por `StreamManager`. Comenzamos en primer lugar por la transmisión de la imagen, la cual seguirá el proceso ejemplificado en la Figura 28.

Para poder determinar el modo más apropiado de realizar la transmisión de la imagen, primero hay que analizar la forma en que Unity nos permite la obtención de la misma. La clase `WebCamTexture` [52] permite el acceso a la webcam del dispositivo en el que se ejecuta la aplicación y la obtención de los cuadros provenientes de la misma. Dichos

cuadros los podemos obtener gracias a los métodos `GetPixels()` o `GetPixels32()` de dicha clase, y con los píxeles obtenidos podemos obtener una textura `Texture2D` que nos permita la representación de la imagen en la pantalla. No obstante, la transmisión directa de un objeto `Texture2D` como campo de un mensaje no es posible ya que las clases derivadas de `MessageBase` únicamente pueden contener campos de tipos sencillos y los tipos más comunes de `UnityEngine` [47]. Por tanto, será necesaria la transmisión directa del vector de píxeles que compone la imagen obtenido a partir de `GetPixels32()` en forma de elementos `Color32`.

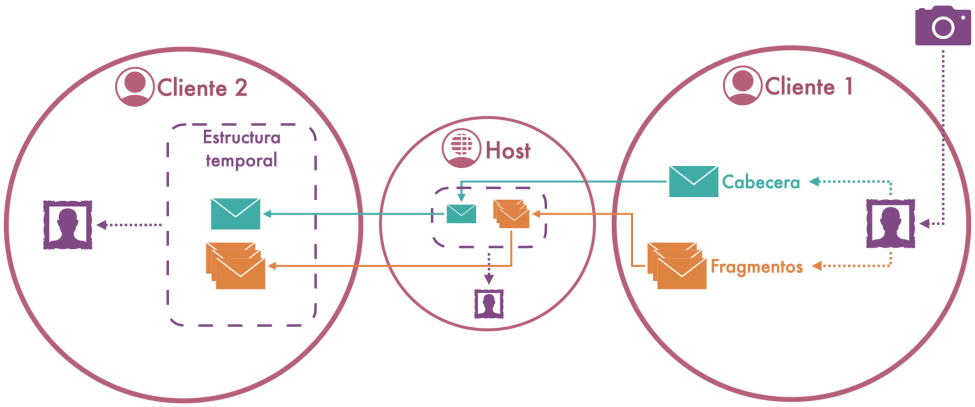

Figura 28: esquema de ejemplo de transmisión de una imagen.

Por tanto, definimos la clase `CamManager` que hereda de `StreamManager`. En esta clase, definiremos los mensajes cabecera y fragmento que emplearemos para la transmisión de la información asociada a imágenes:

- La clase privada `TextureHeaderMessage` que hereda de `StreamHeaderMessage`, incorporando los campos enteros `width` y `height` que nos permitirán la transmisión de la dimensión de la imagen que se transmite. El conocer la dimensión de la imagen en la llegada de la cabecera es importante ya que nos permite conocer el número de píxeles que debemos recibir. La clase define también su constructor vacío obligatorio para el empleo de la clase como mensaje y su propio constructor que toma los argumentos necesarios. También se implementan los métodos `Serialize` y `Deserialize`, los cuales en primer lugar llaman internamente a los métodos `Serialize` y `Deserialize` de `StreamHeaderMessage`.

- La clase privada `TextureChunkMessage` que hereda de `StreamChunkMessage`, incorporando el campo data, el cual es un vector de tipo `Color32` y contendrá el fragmento de píxeles a transmitir. Esta clase también define un constructor vacío, el constructor necesario para inicializar los argumentos y los métodos `Serialize` y `Deserialize`.

- Para la transmisión de los mensajes será necesaria también la definición del tipo que se empleará para los mensajes asociados a imágenes. Por lo tanto, definimos también la clase `TextureMsgType` que hereda de `StreamMsgType`, definiendo un constructor que asigna el valor base a `Header` y `Chunk`. También se encarga de implementar el método `UpdateTypes(…)` que, dado el `netId` del objeto en el que se

incorpora la clase, actualiza los valores de `Header` y `Chunk` para que sean únicos a este objeto.

La clase `CamManager` también define la estructura privada `TextureStruc` que nos permitirá el almacenamiento temporal de las imágenes recibidas conteniendo sus dimensiones y vector de píxeles. Esta estructura podrá ser convertida a `Texture2D` posteriormente para su representación en pantalla. De igual manera, en `CamManager` se definen los campos necesarios para el proceso de captura de la imagen y la transmisión y recepción de la misma, como por ejemplo el objeto `WebCamTexture cam` a utilizar para obtener la imagen, la resolución deseada para la imagen cuadrada que se quiere obtener (en `resolution`), la instancia `textureMsgType` de `TextureMsgType` que define los tipos de los mensajes a emplear o la instancia `msgData` de `StreamMessageDataSupport<TextureStruc, TextureHeaderMessage, TextureChunkMessage>` que se encargará del almacenamiento de la información recibida.

La clase `CamManager` también deberá implementar los métodos necesarios para completar el procesamiento de los mensajes recibidos y la creación de los mismos a partir de las imágenes obtenidas. Para ello, `CamManager` incorpora:

- La implementación de la corrutina `SendStream()`, en la cual se obtiene la imagen a partir de los píxeles deseados obtenidos de `cam`, se aplica un escalado para ajustar la imagen a la resolución deseada y se recorta la imagen para que tenga un relación de aspecto 1:1. Para poder llevar a cabo este escalado, hacemos uso de la función `TextureScale.Bilinear(…)` [39], mientras que para obtener una imagen cuadrada obtenemos solo los píxeles que forman el cuadro central de la imagen con `cam.GetPixels(int x, int y, int blockWidth, int blockHeight)`. Cuando ya tenemos la imagen deseada, construimos un mensaje cabecera a partir de la imagen y los enviamos. A continuación, obtenemos también los píxeles de la imagen y dividimos el vector en fragmentos según el tamaño calculado a partir de `maxChunksize`, enviando un mensaje fragmento por cada uno.

- El método `SaveChunk(TextureChunkMessage row)` que dado un mensaje fragmento, guardará los píxeles contenidos en el mismo en la `TextureStruc` con el mismo identificador almacenado en `msgData`. La posición del fragmento dentro de la imagen se determinará en función del orden del fragmento y el tamaño máximo de los fragmentos.

- El método `OnTextureHeaderReceived(TextureHeaderMessage header)` que se encargará de la creación de una nueva `TextureStruc` a partir de la información de la cabecera que servirá para almacenar los píxeles proporcionados por los fragmentos. Esta estructura será guardada mediante `msgData.RecoverEarlyChunks(…)` que hará uso de `SaveChunk(…)`. Si la marca temporal de la cabecera es correcta según `msdData.CheckTimestamp(…)`, se iniciará la corrutina `msgData.WaitTillReceiveAllTheStream(…)` que esperará mientras no se hayan recibido todos los píxeles.

- El método para el guardado de los fragmentos recibidos `OnTextureChunkReceived(TextureChunkMessage row)`, que hará uso de `SaveChunk(…)`.

- Los administradores de los mensajes recibidos. Los encargados de la recepción en el servidor harán uso de `OnStreamHeaderMessageFromClient(…)` y `OnStreamChunkMessageFromClient(…)`, mientras que los responsables de la recepción en el cliente utilizarán `OnStreamHeaderMessageFromServer(…)` y `OnStreamChunkMessageFromServer(…)`, que a su vez llaman a `OnTextureHeaderReceived(…)` y `OnTextureChunkReceived(…)` respectivamente.

- El método `RegenerateTextureFromReceivedData(unit id)` encargado de obtener de `msgData` la `TextureStruc` asociada al identificador para su conversión en `Texture2D`. Una vez generada la correspondiente `Texture2D`, es guardada como cuadro actual a mostrar.

- Los comandos `CmdStreamIsOn()` y `CmdStreamIsOff()` encargados de la modificación de la variable sincronizada `isStreamOn` en el servidor. También define la RPC `RpcStreamIsOff()` que es llamada por `CmdStreamIsOff()` y que lleva a cabo la limpieza de todos los cuadros recibidos en los clientes.

- La implementación de `StartRecording()` y `StopRecording()`, los cuales se encargarán del inicio y la pausa respectivamente de la webcam del dispositivo, así como también de la comunicación al servidor del encendido o apagado de la transmisión a través de `CmdStreamIsOn()` y `CmdStreamIsOff()`.

- El método `ObtainWebcamImage()` que permite la obtención de la imagen a mostrar actualmente en forma de `Sprite`.

Finalmente, por un lado el método `Start()` de `CamManager` se encargará de la inicialización de `msgData` y `textureMsgType`, así como también de la invocación de `CreateHandlers(…)`. Por otro lado, el método `Update()` se encargará de la ejecución de `BroadcastStream()` cuando la cámara esté encendida, `UpdateStream(…)` y `msgData.ManageUnheadedChunks()`.

Con esto ya podemos llevar a cabo la grabación y transmisión de la cámara de un usuario al añadir el *script CamManager.cs* al *prefab* del avatar del jugador. No obstante, para poder visualizar dicha imagen, es necesario el diseño de otra clase que se encargue de la misma, a la cual denominaremos `CamViewport`. Esta clase será local y contendrá una referencia a un `CamManager`, encargándose de obtener en cada frame el último cuadro de la misma y de su representación mediante un SpriteRenderer.

5.4.5.3.Streaming de audio

El siguiente paso será abordar la transmisión del sonido capturado por el micrófono, siendo los pasos a seguir para su implementación muy similares a los seguidos para `CamManager`. El micrófono puede capturarse gracias a la clase `Microphone` de Unity [50]. Gracias a esta clase, podemos obtener un `AudioClip` que podremos transmitir. No obstante, al igual que con `Texture2D`, la transmisión directa de `AudioClip` no es posible. Por tanto, será necesario obtener el vector de muestras que conforman este sonido para su transmisión mediante el método `GetData(…)` de `AudioClip`.

Por tanto, definimos la clase `MicManager` que hereda de `StreamManager`. En esta clase, definiremos los mensajes cabecera y fragmento que emplearemos para la transmisión de la información asociada a sonidos:

- La clase privada `AudioHeaderMessage` que hereda de `StreamHeaderMessage`, incorporando los campos enteros `samples`, `channels` y `frequency` que representan el número de muestras del sonido, su número de canales y su frecuencia. Estos datos son necesarios para la reconstrucción del audio y determinar el número de muestras que se va a recibir. La clase define también su constructor vacío obligatorio para el empleo de la clase como mensaje y su propio constructor que toma los argumentos necesarios. También se implementan los métodos `Serialize` y `Deserialize`; los cuales, en primer lugar, llaman internamente a los métodos `Serialize` y `Deserialize` de `StreamHeaderMessage`.

- La clase privada `AudioChunkMessage` que hereda de `StreamChunkMessage`, incorporando el campo `data`, el cual es un vector de tipo `float` y contendrá el fragmento de muestras a transmitir. Esta clase también define un constructor vacío, el constructor necesario para inicializar los argumentos y los métodos `Serialize` y `Deserialize`.

- Para la transmisión de los mensajes será necesaria también la definición del tipo que se empleará para los mensajes asociados a sonidos. Por lo tanto, creamos también la clase `AudioMsgType` que hereda de `StreamMsgType`, definiendo un constructor que asigna el valor base a `Header` y `Chunk`, los cuales serán diferentes a los valores base de `CamMsgType`, para evitar la colisión entre los tipos de imagen y sonido. También se encarga de implementar el método `UpdateTypes(…)` que, dado el `netId` del objeto en el que se incorpora la clase, actualiza los valores de `Header` y `Chunk` para que sean únicos a este objeto.

La clase `MicManager` también define la estructura privada `AudioStruc` que nos permitirá el almacenamiento temporal de los sonidos recibidos conteniendo sus características como número de muestras, frecuencia de muestreo, canales y el vector de muestras. Esta estructura podrá ser convertida posteriormente en un `AudioClip` para su reproducción. De igual manera, en `MicManager` se definen los campos necesarios para el proceso de captura del sonido y la transmisión y recepción del mismo, como por ejemplo el objeto `AudioClip` `voiceClip` que recibirá el sonido recogido por el micrófono, la frecuencia de muestreo `frequency` con la que se quiere obtener el sonido, la instancia `clipMsgType` de `AudioMsgType` que define los tipos de los mensajes a emplear o la instancia `msgData` de `StreamMessageDataSupport<AudioStruc, AudioHeaderMessage, AudioChunkMessage>` que se encargará del almacenamiento de la información recibida.

La clase `MicManager` también deberá implementar los métodos necesarios para completar el procesamiento de los mensajes recibidos y la creación de los mismos a partir de los sonidos obtenidos. Para ello, `MicManager` incorpora:

- La implementación de la corrutina `SendStream()`, en la cual se obtienen las muestras contenidas en `voiceClip`, así como el número de canales y muestras del mismo. A partir de estos, construimos un mensaje cabecera y lo enviamos. A

continuación, dividimos el vector de muestras obtenido en fragmentos según el tamaño calculado a partir de `maxChunksize`, enviando un mensaje fragmento por cada uno.

- El método `SaveChunk(AudioChunkMessage row)` que dado un mensaje fragmento, guardará las muestras contenidas en el mismo en la `AudioStruc` con el mismo identificador almacenado en `msgData`. La posición del fragmento dentro del sonido se determinará en función del orden del fragmento y el tamaño máximo de los fragmentos.

- El método `OnAudioHeaderReceived(AudioHeaderMessage header)` que se encargará de la creación de una nueva `AudioStruc` a partir de la información de la cabecera que servirá para almacenar las muestras proporcionados por los fragmentos. Como en `CamManager`, esto se consigue gracias a los métodos `msgData.RecoverEarlyChunks(…)` que hará uso de `SaveChunk(…)`, `msdData.CheckTimestamp(…)` y `msgData.WaitTillReceiveAllTheStream(…)`.

- El método para el guardado de los fragmentos recibidos `OnAudioChunkReceived(AudioChunkMessage row)`, que hará uso de `SaveChunk(…)`.

- Los administradores de los mensajes recibidos al igual que en `CamManager`. Los encargados de la recepción en el cliente utilizarán `OnStreamHeaderMessageFromServer(…)` y `OnStreamChunkMessageFromServer(…)`, que a su vez llaman a `OnAudioHeaderReceived(…)` y `OnAudioChunkReceived(…)` respectivamente.

- El método `RegenerateClipFromReceivedData(unit id)` encargado de obtener de `msgData` la `AudioStruc` asociada al identificador para su conversión en `AudioClip`. Una vez generado el correspondiente `AudioClip`, es guardado como sonido actual a mostrar.

- Los comandos `CmdStreamIsOn()` y `CmdStreamIsOff()` y la RPC `RpcStreamIsOff()` con las mismas funciones que en `CamManager`.

- La implementación de `StartRecording()` y `StopRecording()`, los cuales se encargarán del inicio y la pausa respectivamente del micrófono del dispositivo, así como también de la comunicación al servidor del encendido o apagado de la transmisión a través de `CmdStreamIsOn()` y `CmdStreamIsOff()`.

- El método `ObtainMicrophoneClip()` que permite la obtención del sonido a reproducir actualmente en forma de `AudioClip`.

Finalmente, los método `Start()` y `Update()` llevarán a cabo los mismos pasos que en `CamManager`. El *script MicManager.cs* será también incorporado por el avatar del jugador y un nueva clase `MicPlayer` que tomará una referencia a este `MicManager` y se encargará de la reproducción del `AudioClip` obtenido a partir de `MicManager`. La Figura 29 muestra de forma abreviada la jerarquía de clases que forman `CamManager`, `MicManager` y `StreamManager`.

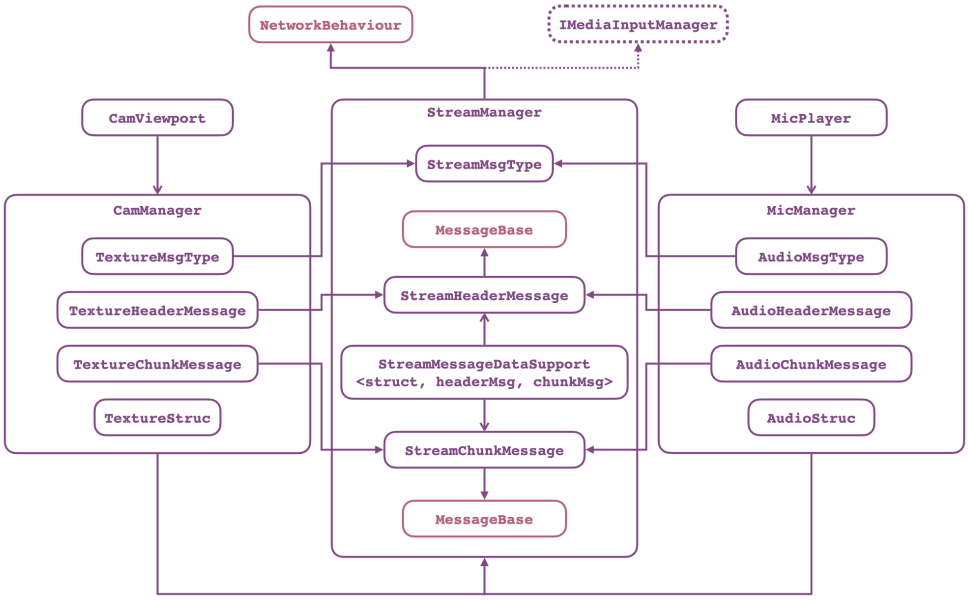

Figura 29: jerarquía de clases para la comunicación audiovisual en la versión UNet.

5.4.5.4. Consideraciones adicionales del streaming

El *prefab* del avatar del jugador también incorporará los componentes `MicPlayer` y `CamViewport` en sus hijos MicSource y CamViewport respectivamente. De esta manera, podemos disponer de una fuente de sonido y de vídeo por cada jugador permitiendo la reproducción simultánea de múltiples micrófonos y cámaras.

Cada usuario podrá decidir cuándo activar o desactivar su micrófono o cámara. Esto es posible gracias a las clases locales `MediaInputController` y `MediaCommunicationManager`. Por un lado, la primera toma la referencia a un objeto `StreamManager`, encargándose de iniciar o detener la transmisión del mismo al pulsar una tecla determinada a través de los métodos definidos en la interfaz `IMediaInputManager`. Dicha clase también se encarga de la gestión de los elementos de interfaz asociados al control de este medio, como son el icono de estado o el texto indicando la tecla de control. Por otro lado, la segunda clase toma una referencia al `MediaInputController` responsable del micrófono y otra referencia al `MediaInputController` responsable del control de la cámara, encargándose de buscar al jugador local en cada cambio de escena para vincular su `CamManager` y `MicManager` con los correspondientes controladores. El lienzo originalmente diseñado para contener el chat se encargará también de alojar los controladores de la cámara y el micrófono, incorporando el componente `MediaCommunicationManager` necesario y pasando a llamarse CommCanvas. La Figura 30 muestra el esquema resumen de los objetos que forman este lienzo.

Una vez terminada la incorporación de la comunicación audiovisual en nuestro juego y antes de terminar esta sección, cabe realizar algunas puntualizaciones:

- El envío de mensajes haciendo uso de la función `SendByChannelToReady(…)` de `NetworkServer` o `SendByChannel(…)` de `NetworkClient` nos permite seleccionar el canal a utilizar para la transmisión. Esto nos permitirá escoger las características de la misma, ya que en el componente NetworkManager es posible

crear nuevos canales. Para ello, activaremos la configuración avanzada del componente y podremos ver la lista de canales *Qos Channels*, la cual podremos editar para añadir un nuevo canal. Podremos escoger el tipo de este canal, el cual determinará si es fiable o no, si garantiza la secuencialidad o si se fragmentan los mensajes. Dependiendo del canal, tendremos unas limitaciones u otras. En nuestro caso, hemos optado por *Reliable Fragmented* ya que evita la restricción de tamaño para los paquetes de 1340 *bytes* de *Reliable*.

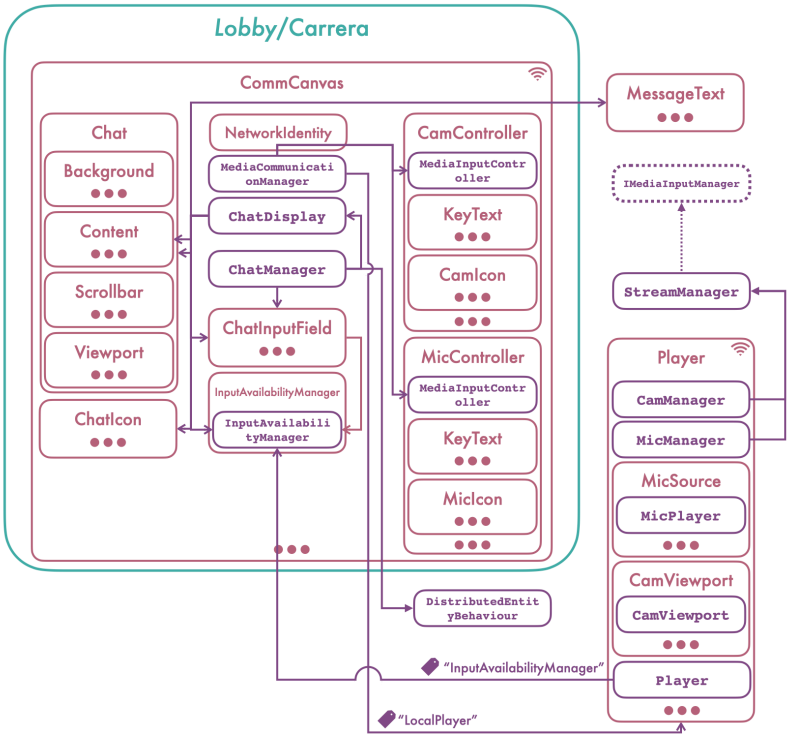

Figura 30: esquema resumen de la estructura de CommCanvas en la versión UNet.

- El número de paquetes que se pueden almacenar por defecto en el *buffer* es de 16. Este número ha probado ser insuficiente para la transmisión de audio e imagen por parte de múltiples usuarios. Dicho valor puede modificarse en el componente NetworkManager como *Max Buffered Packets*, el valor del cual incrementamos a 300. No obstante, dicha modificación no surge efecto sobre las instancias cliente que mantienen su restricción de 16 paquetes. Para que el incremento se aplique en los clientes también, podemos modificar el campo `MaxPendingPacketCount` de la clase `ChannelBuffer` de UNet.

- La clase `Microphone` únicamente permite la captura de fragmentos de sonido cuya longitud es mayor o igual a un segundo. Por este motivo, no tiene sentido llevar cabo más de una transmisión por segundo en el caso de `MicManager`.

- Como se puede ver, los comandos empleados en las clases `CamManager` y `MicManager` son los mismos. Inicialmente, estos comandos se querían incluir en la clase `StreamManager`. No obstante, al definirse en la clase padre, dichos comandos presentaban un comportamiento inesperado por el cual no se ejecutaban en la

instancia servidor. Debido a estos problemas, se optó por su implementación en `MicManager` y `CamManager`.

- La llamada a `DestroyHandlers()` debe invocarse antes de proceder a realizar una desconexión para dar de baja correctamente las funciones asociadas a la recepción de los mensajes. Esto se debe a que en UNet el registro recae directamente sobre el cliente (`NetworkClient`) o sobre el servidor (`NetworkServer`), por lo que la conexión debe estar activa para poder proceder a la dada de baja.

5.5. Transición a MLAPI

La API por la que Unity apuesta para el futuro de los juegos multijugador es, como hemos visto en el capítulo 2, *Mid-Level API* [59][16]. Con el objetivo de explorar la perspectiva de futuro para los juegos multijugador en Unity y las capacidades de este nuevo paquete de desarrollo, decidimos llevar a cabo la transición de nuestro proyecto a MLAPI. Esto nos permitirá poder establecer cuáles son las principales diferencias y similitudes, así como también estudiar cuáles son los pasos necesarios para la adaptación a la nueva propuesta de proyectos que utilicen la HLAPI de UNet y la más reciente actualidad de los juegos multijugador en Unity.

5.5.1. Características de MLAPI

MLAPI se encuentra actualmente en desarrollo, siendo la versión actualmente recomendada para su instalación la v.0.1.0 [16]. No obstante, también es posible acceder a su versiones v.0.1.1 y v.0.2.0 en su página oficial de GitHub [59]. La versión v.0.2.0 es accesible a través de su rama de desarrollo *develop*, por lo que su instalación y uso debe hacerse con precaución debido a su mayor inestabilidad.

La documentación proporcionada por Unity para la versión v.0.1.0 nos permite ver las características que nos ofrece [16]. Al igual que la HLAPI de UNet, MLAPI nos ofrece la posibilidad de establecer conexiones a otros jugadores mediante una nueva versión de la clase `NetworkManager`. No obstante, esta nueva `NetworkManager` no permite la herencia de la misma, lo que se recomendaba en UNet. En nuestro proyecto hemos empleado la clase `NetworkManager` por defecto, por lo que esto no nos supondrá una dificultad adicional, pero debe tenerse en cuenta para aquellos proyectos que hagan uso de una versión personalizada de `NetworkManager`. Durante la conexión con otros jugadores, también dispondremos de objetos distribuidos entre todos los dispositivos gracias a `NetworkObject`, que reemplaza a `NetworkIdentity`, y será posible el cambio de escena de forma sincronizada gracias a `NetworkSceneManager`. Los *scripts* incorporados por los objetos en red seguirán heredando de una nueva versión de `NetworkBehaviour` y para la sincronización de la posición a través de la red contamos con una nueva versión del componente NetworkTransform. La comunicación entre las diferentes instancias de un objeto en red seguirá siendo posible a través de llamadas remotas a funciones mediante las RPC servidor y las RPC cliente, que reemplazan a los comandos y RPC de UNet. Para una comunicación más personalizada, MLAPI cuenta con un servicio de mensajes personalizados mediante `CustomMessagingManager`. La definición de variables sincronizadas será posible a través de la clase `NetworkVariable` que reemplazará al atributo `[SyncVar]` de UNet. Finalmente, MLAPI no ofrece por el momento ningún tipo se sistema de descubrimiento que permita reemplazar la clase `NetworkDiscovery` de UNet,

aunque en la documentación se indica la posibilidad de seguir empleando dicha clase con MLAPI.

Las características presentadas por MLAPI la convierten en un reemplazo viable para la HLAPI de UNet al ofrecer, en la mayoría de los casos, un soporte similar de funciones y clases para el desarrollo de juegos multijugador. Todo esto nos lleva a considerar posible la transición a MLAPI, siendo la falta de un nuevo sistema de descubrimiento la principal funcionalidad ausente en esta librería para nuestro proyecto.

5.5.2.Instalación de MLAPI

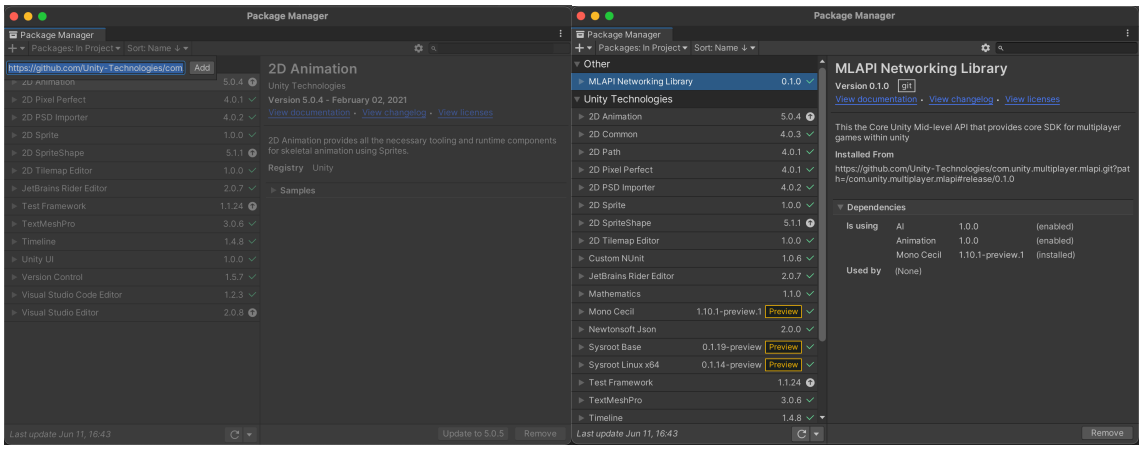

Figura 31: instalación de MLAPI.

La instalación de MLAPI se lleva a cabo de forma diferente a los paquetes tradicionales de Unity, debido a encontrarse todavía en desarrollo y no estar incluida de forma oficial en el registro de Unity. Siguiendo los pasos establecidos en la documentación oficial [18], para proceder a la instalación de MLAPI abriremos la ventana *Package Manager* en el editor de Unity. Es esta ventana, al presionar el símbolo "+" escogeremos la opción *Add package from git URL...* e introduciremos la dirección a la versión v.0.1.0 de MLAPI en GitHub [60]. Con estos pasos ya tendremos instalada la API para su uso y la veremos registrada al igual que en la Figura 31.

5.5.3.Adaptación del proyecto

El uso de MLAPI en nuestro proyecto requerirá la sustitución de todos los componentes de HLAPI de UNet que actualmente dan soporte a nuestro proyecto por sus correspondientes reemplazos en MLAPI. Para este proceso tomamos como referencia la guía de transición de UNet a MLAPI disponible en la documentación oficial [23].

5.5.3.1.Aspectos generales

Uno de los primeros pasos para la transición a MLAPI es el cambio de las clases y componentes empleados. En concreto:

- Todos los objetos que contengan un componente NetworkIdentity deberán reemplazarlo por un componente NetworkObject. Hay que tener en cuenta que NetworkObject no contiene la propiedad *Local Player Authority* debido a que ya no será necesaria para poder ejecutar llamadas remotas al servidor. Además, NetworkObject incorpora una nueva propiedad *Dont Destroy With Owner* que

evitará que el objeto se vea destruido en todos los clientes cuando el jugador que los posea se desconecte o sea destruido.

- El método `Start()` de todos los objetos que hereden de NetworkBehaviour se reemplazará por el método heredado `NetworkStart()`.

- Los comandos deberán hacer uso del atributo `[ServerRpc]` de `MLAPI.Messaging`, siendo necesario renombrar todos los comandos y RPC cliente para que terminen con el sufijo "...ServerRpc" y "...ClientRpc" respectivamente. En MLAPI, podemos conseguir que cualquier instancia ejecute llamadas remotas sobre la instancia servidor al incorporar el argumento `RequiereOwnership`, de la forma `[ServerRpc(RequiereOwnership = false)]`. Esto evita que sea necesario solicitar autoridad al jugador antes de poder ejecutar un llamada remota sobre la instancia servidor, por lo que la clase `DistributedEntityBehaviour` ya no será necesaria. Todas las clases que heredaban de esta pasarán a heredar directamente de la clase `NetworkBehaviour`, pudiendo llamar directamente a las RPC servidor sin necesidad de hacer uso del método `RunCommand(…)`.

- Todas las clases que disponen de sincronización de variables mediante `[SyncVar]` deberán hacer uso de `NetworkVariable`. Para ello, toda variable de tipo `T` etiquetada con `[SyncVar]` eliminará dicha etiqueta y pasará a ser una variable de tipo `NetworkVariable<T>` que instanciaremos con `new NetworkVariable<T>()`. Para modificar el valor de estas variables sincronizadas, bastará con acceder a su propiedad `Value` y llevar a cabo las operaciones necesarias. Hay que tener en cuenta que las modificaciones realizadas sobre el valor de la variable antes de la llamada a `NetworkStart()` no serán replicadas y que, al igual que en UNet, el valor de estas variables únicamente podrá ser escrito por la instancia servidor.

- La instanciación en todos los dispositivos de un *prefab*, como por ejemplo las celdas que componen la pista de la carrera de nuestro proyecto, ya no se realizará mediante `NetworkServer.Spawn(…)`. En su lugar, tras la instanciación de un *prefab*, será necesario acceder a su componente `NetworkObject` sobre el que llamaremos al método `Spawn()`. Como ejemplo, `NetworkServer.Spawn(o)` sería reemplazado por `o.GetComponent<NetworkObject>().Spawn()`.

- El acceso a las propiedades `isServer`, `isClient` e `isLocalPlayer` debe ser reemplazado por el acceso a las propiedades `IsServer`, `IsClient` e `IsLocalPlayer`.

- `NetworkManager.singleton` se sustituye por `NetworkManager.Singleton`.

- Los cambios de escena se llevarán a cabo mediante `NetworkSceneManager.SwitchScene(…)` en lugar de `NetworkManager.ServerChangeScene(…)`.

- Todos los objetos que hagan uso del componente `NetworkTransform` de UNet deberán reemplazarlo por el nuevo componente `NetworkTransform` de MLAPI.

- El componente `NetworkStartPosition` no está disponible en MLAPI, por lo que será necesario eliminar su uso.

- La detección de la desconexión por parte de un cliente se podrá llevar a cabo mediante el registro de una retrollamada asociada al evento `NetworkManager.Singleton.OnClientDisconnectCallback`. Esto reemplaza el seguimiento de los mensajes de tipo `MsgType.Disconnect` de UNet.

- Finalmente, será preciso reemplazar el NetworkManager de UNet por el nuevo NetworkManager de MLAPI. Para el correcto funcionamiento de este NetworkManager, será necesaria la selección del transporte a emplear. En nuestro caso, emplearemos el transporte de UNet ya que es el utilizado por defecto en MLAPI a falta de su reemplazo, aunque cabe destacar que MLAPI ofrece también soporte para utilizar APIs de transporte creadas por el usuario. Será necesario también registrar los *prefabs* a utilizar en este nuevo NetworkManager, así como también las escenas a las que se podrá acceder. Incorporamos por tanto a la lista *Network Prefabs* los *prefabs* de las celdas y del jugador, el cual marcamos como *Default Player Prefab*. En cuanto a las escenas accesibles, registramos en *Registered Scene Names* las escenas "LobbyScene" y "GameScene". Para terminar, permitiremos los cambios de escena al activar las propiedades *Enable Scene Management* y *Allow Runtime Scene Changes*.

Con todo esto ya tenemos las modificaciones básicas que es necesario realizar en nuestro proyecto para comenzar la transición a MLAPI. No obstante, aún tenemos que cubrir algunas dificultades adicionales que surgen de la substitución de UNet y que detallaremos a continuación.

5.5.3.2.Jugador

Una diferencia importante entre HLAPI y MLAPI es que el avatar del jugador no es destruido en el cambio de escena. En su lugar, este es guardado temporalmente mientras se lleva a cabo la transición a la nueva escena y se recupera una vez la escena ha sido cargada. Esto provoca que el método `NetworkStart()` se ejecute en el avatar cuando el jugador entra en el *lobby*, pero no cuando se empieza una carrera. Será por tanto necesario tener en consideración este hecho para conseguir una correcta actualización del estado del avatar del jugador en todo momento. Para detectar el cambio de escena, bastará con suscribirse al evento `OnSceneSwitched` de `NetworkSceneManager`. Al recibir dicho evento, llevaremos a cabo todas las operaciones necesarias para actualizar el estado.

La posición de inicio por defecto para la creación del jugador en MLAPI es el origen de la escena. Para permitir una funcionalidad similar a la proporcionada por `NetworkStartPosition` en UNet, haremos que en cada actualización de estado producida por un cambio de escena, el jugador busque todos los objetos con la etiqueta "SpawnPoint" y se transporte a uno de ellos aleatoriamente. De esta manera, al etiquetar con "SpawnPoint" todos los puntos de inicio que antes incorporaban `NetworkStartPosition` podremos simular su comportamiento.

5.5.3.3.Administración de jugadores y escenas

Uno de los aspectos más importantes del proyecto es el sistema de administración de jugadores implementado en la clase `PlayersManager`, ya que permite la administración del proceso de separación de jugadores cuando se inicia una nueva partida, asegurando que haya un host para los jugadores del *lobby* y otro para los jugadores de la carrera. Garantizar la conservación de su comportamiento original es

fundamental. Por ello, es necesario tener en cuenta las siguientes consideraciones en su adaptación a MLAPI:

- MLAPI asigna a cada jugador con el que se está conectado un identificador, el cual permite tanto la identificación del cliente como su conexión con el mismo. En nuestra implementación actual, empleamos el `netId` del avatar del jugador para identificar a los diferentes clientes. En nuestra nueva versión, utilizaremos en su lugar el identificador del cliente, el cual obtendremos mediante `NetworkManager.LocalClientId` y enviaremos al servidor mediante una RPC servidor.

- El registro de los jugadores y la inicialización se llevaban a cabo en UNet mediante `OnStartClient()` y `OnStartServer()`. No obstante, es MLAPI dichas funciones no existen, por lo que implementaremos su funcionalidad en el nuevo `NetworkStart()`.

- Para llevar a cabo el nombramiento de un nuevo host o la redirección de un cliente, en UNet era necesario el envío de una RPC a todos los clientes y que estos comprobaran si eran el destinatario del mismo. Con MLAPI, podemos hacer uso de `ClientRpcParams` para especificar el cliente (o clientes) que debe recibir la llamada [7]. Estableciendo la propiedad `TargetClientIds` de una instancia de `ClientRpcParams` y pasando la misma como argumento de la RPC, podemos enviar únicamente al destinatario deseado las señales de nuevo host o la redirección de conexión.

- La creación de un nuevo host se llevará a cabo de forma similar a la versión de UNet. No obstante, cabe hacer dos apreciaciones. La primera, para seleccionar el puerto sobre el que abrir el nuevo host hay que acceder al componente `UNetTransport` de `NetworkManager.Singleton`, y sobre él modificar el valor de `ServerListenPort`. La segunda, entre el cierre del actual cliente y la creación del nuevo servidor será preciso esperar a que el avatar del jugador local sea destruido, ya que si no se lleva a cabo esta espera, la eliminación no se produce de forma adecuada causando una destrucción indeseada del objeto en el cambio de escena.

- La redirección de un cliente hacia un nuevo host requerirá también, por precaución, la espera a la destrucción del jugador local entre la desconexión y la reconexión del cliente. Para establecer el puerto y la dirección de la conexión será necesario acceder al componente `UNetTransport` de `NetworkManager.Singleton`, y sobre él modificar el valor de `ConnectPort` y `ConnectAddress` respectivamente. Cabe destacar la diferenciación de MLAPI entre `ServerListenPort`, el puerto sobre el que se inicia el host, y `ConnectPort`, el puerto al que se conecta el cliente, ya que esta diferenciación no está presente en UNet.

- Tanto la creación de un nuevo host como la redirección de un cliente se llevarán a cabo en una corrutina en lugar de en un método. Esto permite que la RPC termine tras el inicio de la corrutina antes del cambio de conexión, lo que evita la generación de errores inesperados.

- El cierre del host actual se llevará a cabo de forma muy similar, aunque para acceder al número de jugadores conectados será necesario el uso de `NetworkManager.Singleton.ConnectedClients.Count`. No obstante, el

cierre de un cliente requiere en MLAPI un proceso diferente. Anteriormente debíamos ejecutar un comando para la eliminación de la información asociada a este cliente, para lo que era necesario solicitar la autoridad necesaria y esperar hasta que esta autoridad fuera revocada tras la ejecución del comando para cerrar el cliente. Con la nueva capacidad que ofrece MLAPI para ejecutar una RPC servidor sin solicitar autoridad, podemos llevar a cabo de forma directa la RPC. Sin embargo, deberemos asegurarnos de que la RPC ha sido correctamente ejecutada antes de cerrar el cliente ya que si no, podríamos interrumpir la conexión antes de que la RPC se pueda invocar. Por ello, creamos una nueva variable booleana `waitingCloseSignal` que nos ayudará en el proceso de cierre de un cliente. De esta forma, tras la llamada a la RPC servidor, el cliente esperará mientras `waitingCloseSignal` sea falsa y será el servidor el encargado de llamar una RPC cliente que modifique dicha variable a verdadero únicamente para este cliente, notificando así la correcta ejecución de la RPC servidor. Tras la notificación por parte del servidor, solo quedará cerrar el cliente.

- `PlayersManger` lleva a cabo el registro de la dirección IP de los clientes conectados para permitir la posterior redirección cuando sea necesario nombrar a un nuevo host. Esta IP se determina en UNet gracias al acceso a la propiedad `connectionToClient.address` del `NetworkIdentity` de la instancia servidor del avatar del jugador. No obstante, esta propiedad no está disponible en MLAPI. Por tanto, para poder obtener la IP del cliente, haremos uso del método `GetUNetConnectionDetails(…)` del componente `UNetTransport` de `NetworkManager.Singleton`. Este método nos permite obtener el identificador de la conexión y del host a partir del identificador del cliente. Con estos identificadores, podemos obtener la dirección IP del cliente, entre otros parámetros de la conexión, mediante el método `GetConnectionInfo(…)` de `UnityEngine.Networking.NetworkTransport`.

Otro aspecto a tener en cuenta es la limitación de MLAPI en la conservación de objetos entre escenas. En nuestro proyecto de UNet, los objetos PlayersManager y CommCavas hacen uso de `DontDestrotOnLoad(…)` para persistir durante el cambio de escenas. No obstante, a pesar del uso de este método, MLAPI no permite la persistencia de objetos entre escenas [30]. Por tanto, optamos por la creación en la escena GameScene de una copia de PlayersManager y CommCanvas. Esto presenta el inconveniente de la falta de persistencia del estado de estos objetos, que se conseguía con `DontDestroyOnLoad(…)`, reseteándose el estado por defecto en el cambio a GameScene. Esto es más notable en CommManager, donde la cámara y el micrófono se apagarán en el cambio a GameScene.

5.5.3.4.Descubrimiento de jugadores

Nuestro proyecto hace uso del sistema `NetworkDiscovery` de UNet para el descubrimiento de jugadores en la red local, permitiendo hacer transparente para el jugador los detalles de la conexión como direcciones IP, puertos y la inicialización explícita de un host o un cliente. No obstante, como hemos mencionado en el punto 5.5.1, MLAPI no proporciona por el momento ningún reemplazo para el `NetworkDiscovery` de UNet, aunque se especifica la posibilidad de emplear este con MLAPI.

Sin embargo, la instalación simultánea de la HLAPI de UNet y MLAPI conlleva un conflicto de paquetes que provoca errores en el proyecto, haciendo imposible la compilación del mismo. Esto se debe a que MLAPI tiene como dependencia la versión 1.10.1-preview.1 de Mono Cecil, mientras que Multiplayer HLAPI tiene como dependencia la versión 1.10.1 de Mono Cecil. Dicha doble instalación de diferentes versiones de Mono Cecil es la que provoca conflictos y errores en el proyecto, impidiendo que el proyecto pueda ser compilado como puede verse en la Figura 32.

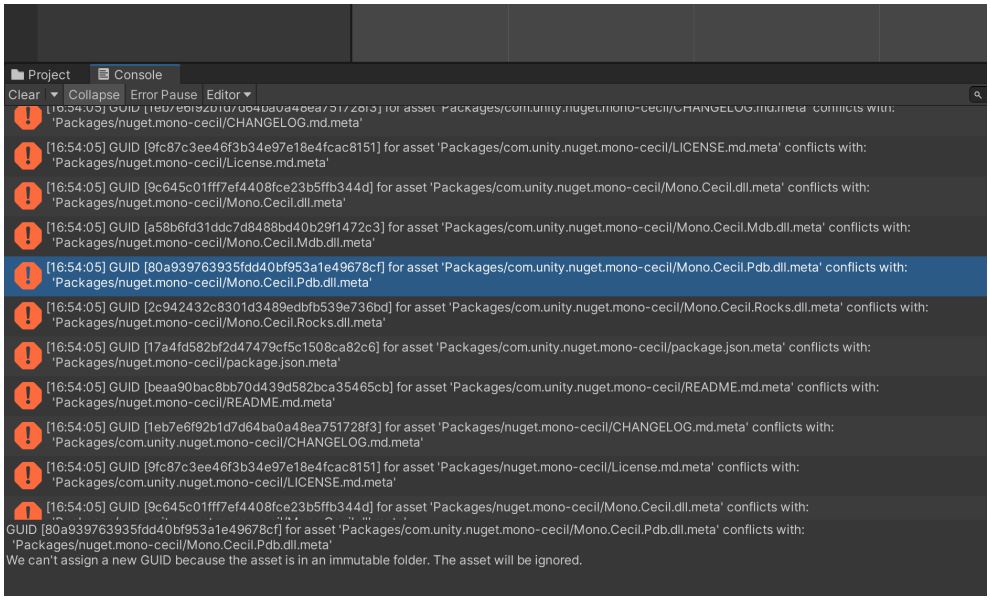

Figura 32: captura de los errores causados por el conflicto de instalación entre UNet y MLAPI.

Para tratar de solucionar el problema y evitar tener instalados tanto el paquete de MLAPI como de Multiplayer HLAPI, decidimos tratar de adaptar el código original de la clase `NetworkDiscovery` a MLAPI. Para ello, primero buscamos en nuestro paquete modificado de HLAPI el código *NetworkDiscover.cs*, lo copiamos dentro de la carpeta *Scripts/Network* del proyecto, y procedimos a la eliminación del paquete Multiplayer HLAPI. A continuación, cambiamos el espacio de `NetworkDiscovery` de `UnityEngine.Networking` a `MLAPI` y procedimos a la identificación de todas las líneas conflictivas con la nueva librería:

- Las líneas relativas a la escritura de mensajes y alertas en el registro de depuración por la ausencia de `LogFilter`. Para reemplazarlo, definimos la variable booleana privada `m_ShowDebugMessages` que permite la activación o desactivación de los mensajes de depuración de `NetworkDiscovery`.

- Las referencias a `NetworkManager.activeTransport`, que han sido reemplazadas por `UnityEngine.Networking.NetworkTransport`.

- Las referencias a `NetworkTransport`, que han sido sustituidas por `UnityEngine.Networking.NetworkTransport`.

- La obtención del puerto de conexión y dirección a partir de `NetworkManager`. Su obtención se ha actualizado para hacer uso de `NetworkManager.Singleton.GetComponent<UNetTransport>().ConnectAddress` y

```
NetworkManager.Singleton.GetComponent<UNetTransport>().ConnectPort.
```

- Las referencias a `NetworkManager.singleton`, que han sido reemplazadas por `NetworkManager.Singleton`.

Con estas modificaciones ya podemos hacer uso del sistema `NetworkDiscovery` de igual forma que en la versión de UNet. Únicamente será necesaria la actualización del módulo en `PlayersFinder` con los mismos parámetros que la versión anterior.

5.5.3.5.Comunicación multimedia

La comunicación entre jugadores a través de la cámara y el micrófono del dispositivo, como hemos visto en el apartado 5.4.5, se consigue mediante el uso de mensajes personalizados que permiten la transmisión de imágenes y fragmentos de sonido. Estos mensajes se basan en la derivación de la clase `MessageBase` de HLAPI y en el empleo de las funciones proporcionadas para la recepción y envío de este tipo de mensajes derivados de `MessageBase`. No obstante, MLAPI no proporciona ninguna clase equivalente a `MessageBase`, y su sistema de mensajes personalizados se basa en el envío y recepción directo de cadenas de *bytes* mediante `CustomMessagingManager`. Será por tanto necesario revisar la implementación de nuestro modelo de retransmisión.

La clase `CustomMessagingManager` permite el envío y la recepción de dos tipos de mensajes: no-etiquetados (*unnamed*) y etiquetados (*named*) [10]. Los primeros permiten la comunicación a través de un canal, pero no permiten la diferenciación de distintos tipos de mensajes. Por otro lado, los segundos permiten la distinción de diferentes tipos de mensajes y la recepción separada de los diferentes tipos registrados. Por tanto, escogemos para nuestro sistema de transmisión hacer uso de mensajes etiquetados, ya que nos permitirán distinguir los diferentes tipos de mensajes cabecera y fragmento de igual forma que en nuestra implementación original.

El primer paso será, por tanto, la modificación de la clase `StreamManager`. En dicha clase, primero modificaremos la clase `StreamMsgType` para que sus campos `Header` y `Chunk` sean cadenas en lugar de enteros, ya que `CustomMessagingManager` emplea cadenas de caracteres para diferenciar el tipo de los mensajes que se envían y reciben. A continuación, debemos llevar a cabo la modificación de las clases `StreamHeaderMessage` y `StreamChunkMessage`, ya que no pueden heredar de `MessageBase`.

La transmisión de mensajes a través de `CustomMessagingManager` se basa en el uso directo de cadenas de *bytes* a transmitir. Por ello, no será posible el envío/recepción directo de `StreamHeaderMessage` y `StreamChunkMessage`. En su lugar, deberemos dotar a estas clases de mecanismos que permitan fácilmente la conversión de las mismas en cadenas de *bytes* y viceversa. Para ello, creamos una nueva clase abstracta `StreamMessage`. Esta clase contiene los campos `netId` e `id`, ya que son comunes a `StreamHeaderMessage` y `StreamChunkMessage`, y define la propiedad abstracta de lectura `MessageStream`, que permitirá obtener la cadena de *bytes* que codifica los campos del mensaje. Así, las clases `StreamHeaderMessage` y `StreamChunkMessage` heredan de `StreamMessage` e implementan la propiedad `MessageStream`. Para la implementación de dicha propiedad instanciamos un nuevo

`MemoryStream` y hacemos uso de la clase `PooledNetworkWriter` de `MLAPI.Serialization.Pooled` para la escritura en dicho `MemoryStream` de los campos de la clase, llevando a cabo esta escritura como en el método `Serialize(…)` de la implementación en UNet. De igual forma, creamos un nuevo constructor para ambas clases que tomará como argumento una cadena de *bytes* e iniciará los campos según su contenido. Para ello, este constructor hace uso de la clase `PooledNetworkReader` de `MLAPI.Serialization.Pooled`, llevando a cabo la lectura de la cadena de *bytes* como en el método `Deserialize(…)` de la implementación en UNet.

Una vez preparadas las clases que modelan la base para los mensajes a transmitir, será necesario proceder a la modificación de las clases encargadas de la recepción y el envío de los mensajes. Empezamos con la función `CreateHandlers(…)`. Dicha función tomará como argumentos un objeto `StreamMsgType` y cuatro funciones de tipo `CustomMessagingManager.HandleNamedMessageDelegate`, en lugar de `NetworkMessageDelegate`, y se encargará de registrar los tipos de mensajes a recibir asignando las funciones proporcionadas como receptores de dichos tipos de mensajes. Para ello, hará uso del método `RegisterNamedMessageHandler(…)` de `CustomMessagingManager`. Una diferencia importante con respecto a UNet es que el registro para la recepción de mensajes se lleva a cabo en la misma clase tanto para instancias cliente como para instancias servidor. Por tanto, para diferenciar los mensajes dirigidos a un servidor y los dirigidos a un cliente, será necesario crear tipos adicionales de mensajes según vayan destinados al servidor o al cliente. Esta diferenciación se producirá únicamente de forma interna en `StreamManager`, por lo que para ello se añadirá a los tipos registrados el sufijo “Server” para la instancia servidor y el sufijo “Client” para la instancia cliente. Se modifica también `DestroyHandlers(…)` para hacer uso del método `UnregisterNamedMessageHandler(…)` de `CustomMessagingManager`.

Los siguientes métodos que será preciso modificar son los encargados del envío de mensajes. Antes de proceder con dicha modificación, es necesario aclarar el funcionamiento del sistema de envío de `CustomMessagingManager`. A través del método `SendNamedMessage(…)` podemos proceder al envío al cliente indicado de la cadena de *bytes* asociada a un tipo concreto de mensajes. No obstante, el envío de mensajes a sí mismo no es posible en este sistema, no permitiendo el envío interno dentro del host de un mensaje personalizado desde la instancia servidor a su instancia cliente interna. Esto es un detalle importante ya que nuestra implementación en UNet se basa en dicha transmisión para la recepción de las imágenes y sonidos en la instancia host, por lo que deberemos tenerlo en cuenta en la adaptación de `CamManager` y `MicManager`. Por el momento, cuando se envíe un mensaje desde el servidor, este será enviado uno a uno a todos los clientes conectados excepto a sí mismo, añadiendo al tipo de mensaje enviado el sufijo “Client” ya que los mensajes serán recibidos por las instancias cliente. Finalmente, los mensajes enviados por una instancia cliente se enviarán al servidor gracias al identificador `NetworkManager.Singleton.ServerClientId`, incorporando el sufijo “Server” a su tipo. El servidor se encargará de su retransmisión a todos los clientes ya que es el único que cuenta con la lista completa de clientes conectados [25].

A continuación, deberemos proceder a la modificación de `CamManager` y `MicManager`. Será necesario modificar las clases `TextureMsgType` y `AudioMsgType` para que hagan uso de cadenas de caracteres para los campos `Header` y `Chunk`, garantizando que permitan diferenciar los mensajes de audio e imagen. También será necesaria la modificación de `TextureHeaderMessage`, `TextureChunkMessage`, `AudioHeaderMessage` y `AudioChunkMessage` para la implementación de la propiedad `MessageStream`, en la cual se obtendrá la cadena de *bytes* procedente de `base.MessageStream` y se ampliará su información con los campos incorporados por estas clases. De igual manera, se modifican estas clases para la incorporación de un constructor que tome como argumento una cadena de *bytes*, se encargue de llamar al constructor base correspondiente e inicialice los campos incorporados a partir de la lectura de la cadena.

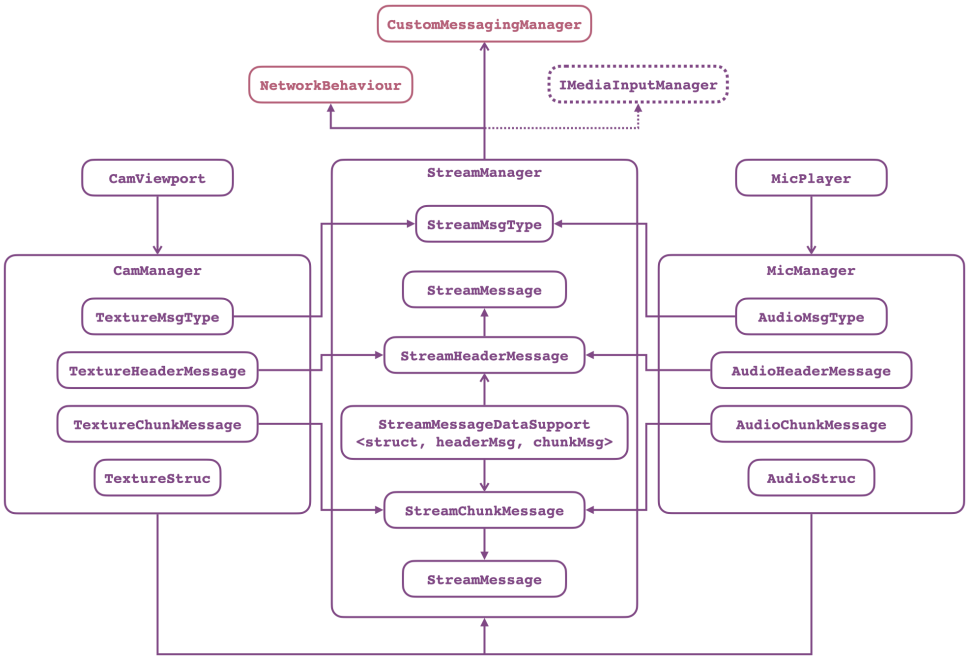

Figura 33: jerarquía de clases para la comunicación audiovisual en la versión MLAPI.

Será necesario también modificar en `CamManager` y en `MicManager` las funciones encargadas de recibir los mensajes para que tomen como argumento un *ulong* y una cadena de *bytes* en lugar de un objeto `NetworkMessage`. La implementación de las funciones encargadas de recibir mensajes en los clientes será la misma a excepción de la regeneración del mensaje a partir de la cadena de *bytes*. En cuanto a la recepción de mensajes en el servidor, será preciso que la función llame tanto al método `OnStreamHeaderMessageFromClient(…)` / `OnStreamChunkMessageFromClient(…)` como al método `OnStreamHeaderMessageFromServer(…)` / `OnStreamChunkMessageFromServer(…)`, ya que así simularemos el envío interno del mensaje recibido en el servidor. Finalmente, modificamos `SendStream()` para que cuando sea la instancia host la que capture la imagen o el sonido, se guarden directamente los datos ya que no serán enviados a sí misma. La Figura 33 muestra de

forma abreviada la jerarquía de clases que forman `CamManager`, `MicManager` y `StreamManager` al adaptarse a MLAPI.

La última adaptación necesaria para conservar la funcionalidad de UNet reside en la incorporación a `IMediaInputManager` de la propiedad pública booleana `IsOn` y su implementación en `StreamManager`. Esta propiedad nos permitirá saber si se está transmitiendo contenido, lo cual podremos utilizar para recuperar la transmisión en el cambio del *lobby* a la carrera, evitando que la cámara y el micrófono se apaguen al cargar GameScene. Esto es necesario debido a la destrucción y regeneración de CommCanvas en la transición de escena que reinicia el control sobre la transmisión.

5.5.3.6.Últimas consideraciones

Durante el proceso de transición a MLAPI, han surgido dificultades y errores que cabe destacar. En primer lugar, al principio el cambio de escena producía un error de sincronización de objetos en los clientes. Tras una búsqueda, pudimos encontrar que este era un error que ya había sido reportado [31]. Como solución al problema se planteaba la instalación de la rama de desarrollo, ya que en esta rama ya se había solucionado. Por este motivo, decidimos proceder, en primer lugar por precaución, a la instalación de la versión 0.1.1 de MLAPI para comprobar si esta ya solucionaba el error, ya que la versión de la rama de desarrollo se advertía como inestable. Esta instalación puede llevarse a cabo, o bien introduciendo la dirección de GitHub de la versión 0.1.1 de igual forma que en el apartado 5.5.2, o bien mediante la descarga manual del paquete desde GitHub y su integración en la carpeta *Packages* de Unity, procediendo de igual forma que con la incorporación directa en el proyecto del código de UNet. Finalmente, la instalación de la versión 0.1.1 solucionó este problema.

Otra dificultad que nos hemos encontrado afectaba a la generación procedural del terreno, ya que algunas celdas consecutivas que se generaban correctamente en el host no eran correctamente sincronizadas en los clientes. Finalmente, pudimos identificar que esto se debía a que al generar las celdas de golpe al inicio de la partida se producía un desbordamiento del *buffer* de mensajes a transmitir de MLAPI, haciendo que los mensajes asociados a la sincronización de algunas celdas no fueran enviados. Este problema fue posible resolverlo al incrementar el valor de *Max Sent Message Queue Size* de NetworkManager y al ralentizar la generación de nuevas celdas mediante la espera arbitraria de 0,01 segundos entre celda y celda. Así, convertimos el proceso de generación en una corrutina para conceder a MLAPI un mayor tiempo para enviar los mensajes correspondientes.

Para terminar, el último problema con el que nos encontramos es el hecho de que la sincronización de la posición de CellGenerator no siempre se refleja en los clientes al inicio de partida, no sincronizándose hasta que se genera una nueva celda. Dicho problema también ha sido reportado a los desarrolladores, estando pendiente para ser reparado [32]. Lamentablemente, no hemos podido encontrar ninguna solución temporal a dicho problema hasta que los desarrolladores puedan solucionarlo.

Las Figuras 34, 35, 36 y 37 muestran los esquemas resumen de las modificaciones llevadas a cabo para adaptar el proyecto a MLAPI. En las Figuras 35 y 36 se han excluido las relaciones de elementos como `CommChat` y `ScoresManager` ya que estas se muestran en las Figuras 34 y 37 respectivamente. De igual modo, algunos elementos comunes tanto a la Figura 35 como a la Figura 36 han sido omitidos en la segunda con la intención de mejorar la legibilidad.

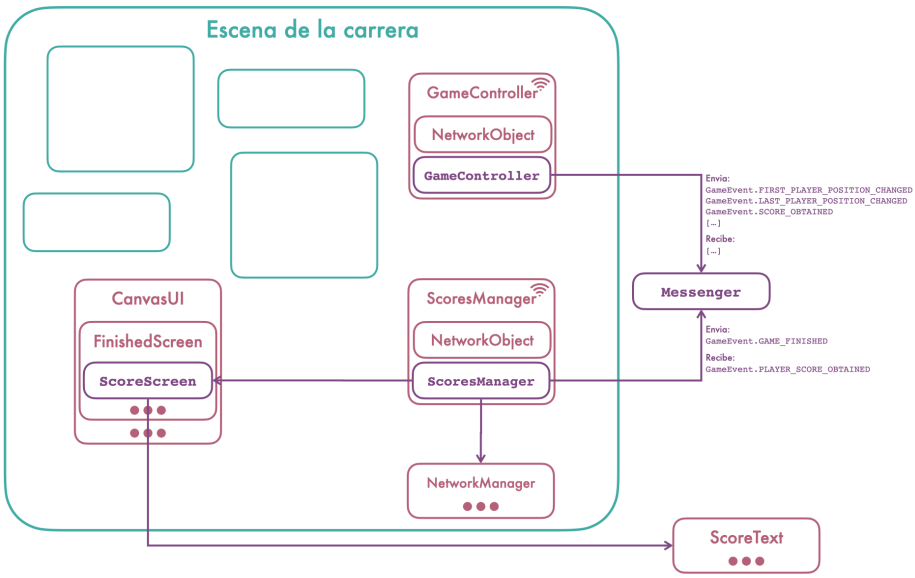

Figura 34: esquema resumen de las relaciones de ScoresManager con otros objetos en la versión MLAPI.

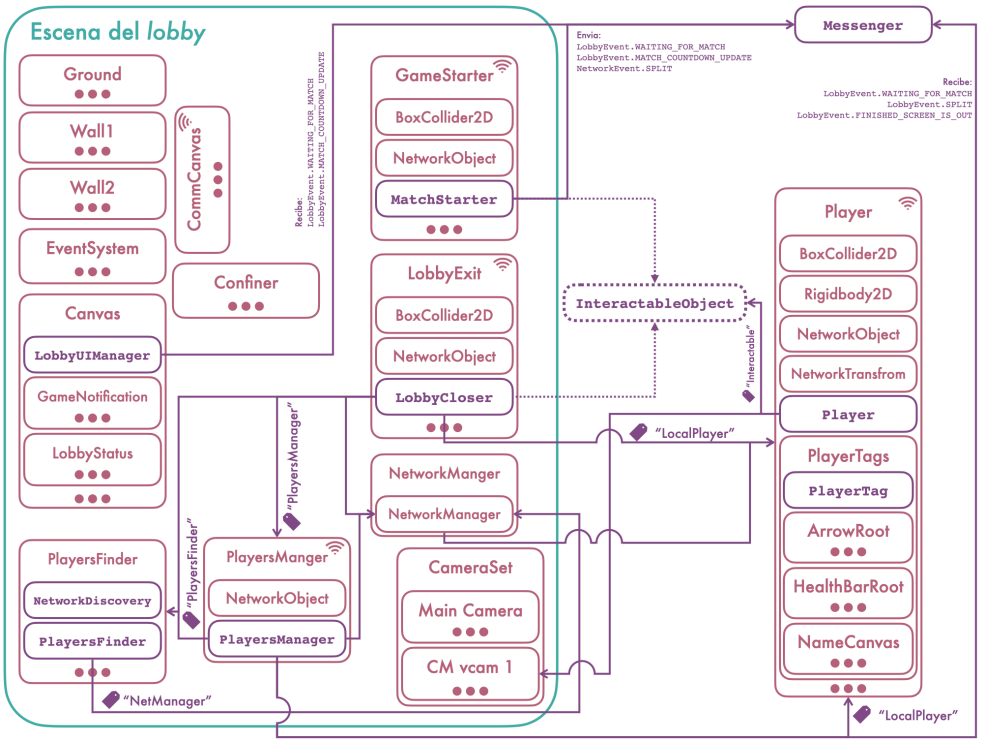

Figura 35: esquema resumen de los objetos que conforman el *lobby* en la versión MLAPI.

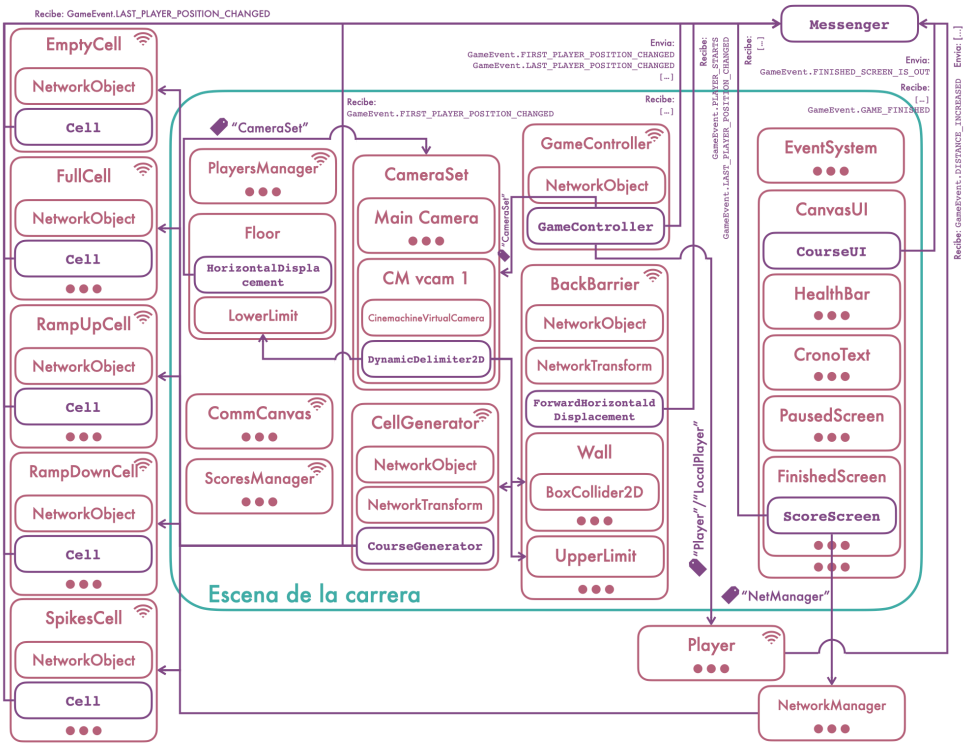

Figura 36: esquema resumen de los objetos que conforman la escena carrera en la versión MLAPI.

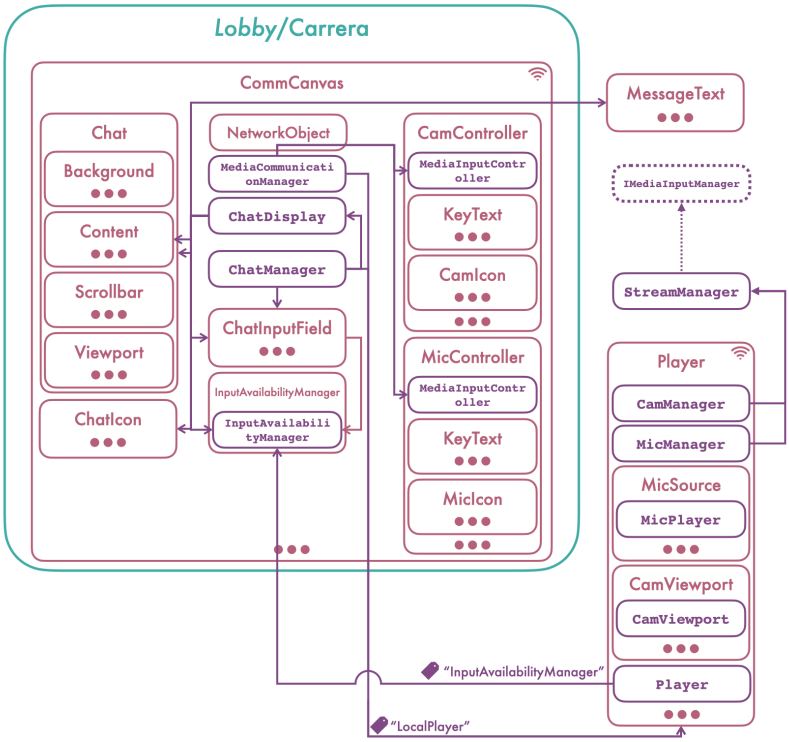

Figura 37: esquema resumen de la estructura de CommCanvas en la versión MLAPI.

5.6.Mejoras generales

Finalmente, en esta sección queremos presentar brevemente algunas modificaciones efectuadas sobre el juego de forma progresiva durante el desarrollo de sus diferentes fases. Estas mejoras cubren principalmente cambios estéticos para mejorar la apariencia del juego y mejoras de calidad de vida. Estas modificaciones son:

- La sustitución de los *sprites* originales y la modificación de la estética de la interfaz para dar al juego un aspecto más acabado y consistente a lo largo de toda la experiencia. Todos los *sprites* y *assets* empleados para ello son propios, a excepción del fondo, el cual obtuvimos gratuitamente del *Unity Asset Store* [15], y las fuentes empleadas para el texto, que fueron descargadas de forma gratuita de *Font Space* [14] [21].

- La incorporación de un sencillo menú de ajustes accesible desde la esquina superior derecha del menú principal, cuya captura se muestra en la Figura 38. Este menú de configuración permite la selección de la cámara y el micrófono a utilizar, así como el número de cuadros por segundo de la cámara y la frecuencia de muestreo del micrófono.

- La incorporación de la posibilidad de interactuar con los iconos del lienzo de comunicaciones mediante el ratón, de tal forma que al hacer clic encima de ellos se pueda activar o desactivar la cámara, el micrófono o el chat. A su vez, incorporamos la posibilidad de interactuar rápidamente con el menú principal al presionar las teclas intro para jugar o escape para cerrar la aplicación. De esta manera, damos un mayor soporte a diferentes estilos de interacción, cubriendo tanto las expectativas de un público más acostumbrado a los atajos de teclado como las de un público más casual acostumbrado a la interacción mediante el ratón. La Tabla 3 muestra este nuevo esquema de controles.

- La adición de un pequeño texto en la parte inferior del lienzo de comunicación que indica el nombre del jugador local y si está llevando a cabo el papel de host o cliente. Esta característica está orientada a facilitar la demostración del juego, visibilizando mejor la relación entre las diferentes instancias del juego y cómo interactúan.

Controles	
Tipo de interacción	Método de interacción
Movimiento del jugador.	Flechas izquierda / derecha. Teclas A / D.
Interacción con las puertas.	Tecla Z.
Salto.	Barra espaciadora.
Pausar / reanudar durante la carrera.	Tabulador.
Volver a *lobby* tras la carrera.	Intro.

Controles	
Tipo de interacción	Método de interacción
Interacción con el menú principal.	Ratón. Intro para jugar. Escape para salir.
Interacción con la interfaz de comunicación.	Ratón. Tecla C para activar / desactivar la cámara. Tecla V para activar / desactivar el micrófono. Tecla M para mostrar / ocultar el chat. Intro para enviar el mensaje escrito.

Tabla 3: esquema de controles de la versión final del juego.

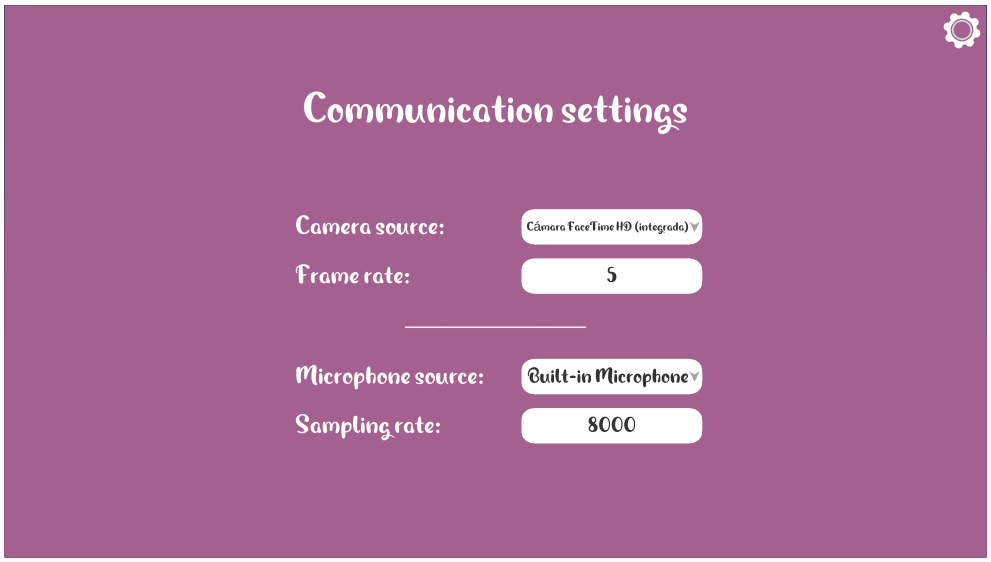

Figura 38: captura del menú de ajustes.

6.Resultados

Una vez terminado nuestro proyecto, cabe presentar los resultados que hemos obtenido. Tras el desarrollo, contamos actualmente con dos versiones en red del proyecto. Por un lado, una versión cuya implementación se ha llevado a cabo sobre la HLAPI de UNet y, por otro, una versión cuya implementación se ha desarrollado sobre MLAPI.

Ambas versiones poseen, esencialmente, la misma funcionalidad y presentan el mismo juego, el cual puede verse en las Figuras 39, 40, 41 y 42. Ambas versiones cumplen con los requisitos de tiempo real que implica la categoría L3 a la que pertenece nuestro juego, aunque los retrasos pueden llegar a ser apreciables, principalmente en el movimiento de los jugadores.

Figura 39: captura del menú principal del juego.

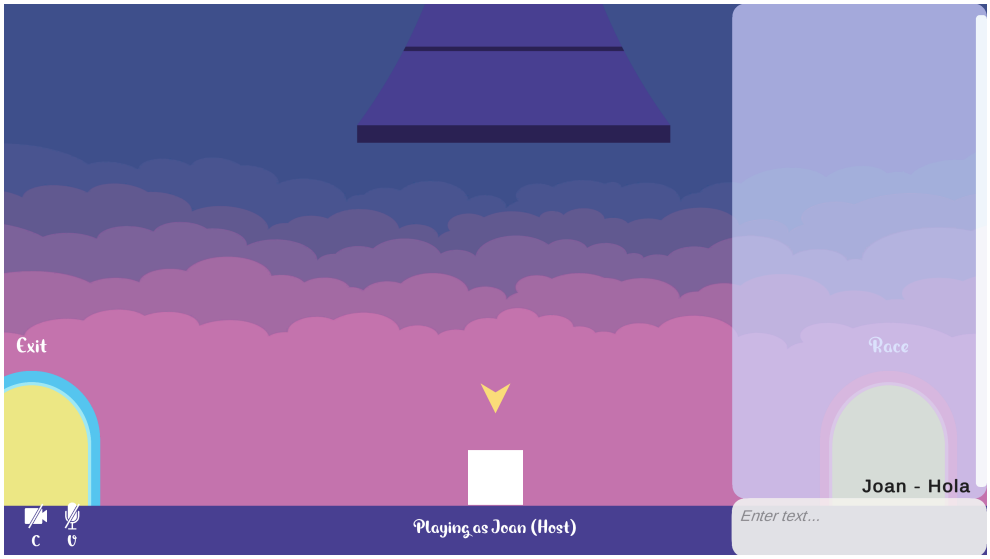

Figura 40: captura de la sala de *lobby* con el chat abierto.

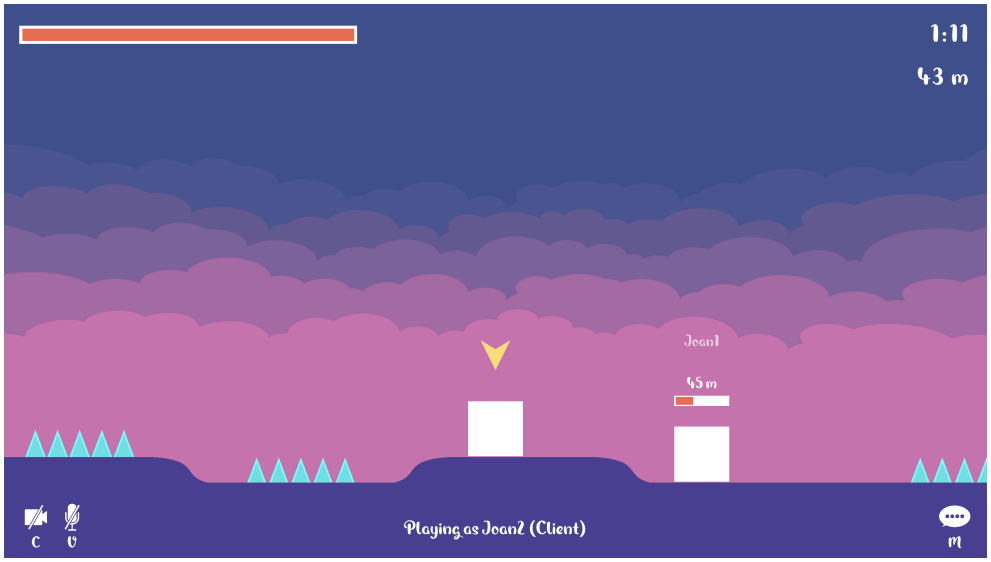

Figura 41: captura de la carrera entre dos jugadores.

Figura 42: captura de la tabla de resultados de una carrera.

El proceso de emparejamiento permite la formación de la topología cliente-servidor host entre todas las instancias del juego ubicadas dentro de la misma red local. No obstante, la naturaleza aleatoria del tiempo de espera del sistema de descubrimiento ha probado no ser suficiente para evitar, en algunos casos, la creación de dos host independientes cuando dos instancias del juego se inician de forma simultánea.

La comunicación audiovisual entre los jugadores también ha probado ser posible en ambas versiones como puede verse en la Figura 44. Sin embargo, la transmisión de la imagen y el sonido de forma simultánea ha mostrado tener efectos sobre la experiencia de juego y la efectividad de la conexión. Para evaluar este fenómeno, llevamos a cabo para ambas versiones un experimento consistente en variar tanto el número de imágenes transmitidas como la frecuencia de muestreo del audio a transmitir y comprobar cuántos jugadores podían mantener una conexión simultánea sin experimentar efectos adversos. Para ello, todas las pruebas han consistido en la conexión de clientes con la cámara encendida a un host con micrófono y cámara

encendidos, estando todas las instancias en la misma máquina. Los motivos de este procedimiento residen en que el host es el que tiene más información que transmitir ya que es el puente entre todos los clientes, lo que hace que sea la instancia más crítica y la que refleja fácilmente la saturación de la conexión. A la hora de determinar el número máximo de clientes, evaluamos la calidad del sonido recibido en los mismos por parte del micrófono del host, la calidad de la sincronización del movimiento y la sincronización de la cámara. La Tabla 4 junto con la Figura 43 presentan los resultados obtenidos y el fenómeno adverso que se produce con el número de jugadores indicado. De la tabla y la figura, como cabe esperar, vemos que cuantas menos imágenes y muestras sea necesario enviar, más jugadores podrán conectarse sin sufrir efectos adversos. En ambas versiones, el principal motivo de estos efectos es el desbordamiento del *buffer* de mensajes de salida, el cual se ha establecido a un tamaño de 300 mensajes para las dos versiones.

Imágenes transmitidas por segundo	Muestras transmitidas cada segundo	Número de jugadores	
		Versión en UNet	Versión en MLAPI
30	44100	2 Pérdida de la transmisión de sonido.	2 Pérdida de la transmisión de sonido.
30	32000	2 Pérdida de la transmisión de sonido.	2 Pérdida de la transmisión de sonido.
30	16000	2 Pérdida de la transmisión de sonido.	2 Pérdida de la transmisión de sonido.
30	8000	2 Muy elevado retraso en la transmisión del sonido hasta su eventual pérdida.	3 Pérdida de sonido y altos retrasos en la sincronización. Los retrasos comienzan a ser notables con dos jugadores.
20	44100	2 Pérdida de la transmisión de sonido.	2 Pérdida de la transmisión de sonido.

Imágenes transmitidas por segundo	Muestras transmitidas cada segundo	Número de jugadores	
		Versión en UNet	Versión en MLAPI
20	32000	3 Pérdida de sonido. Los retrasos comienzan a ser notables con dos jugadores.	2 Pérdida de la transmisión de sonido.
20	16000	2 Pérdida de la transmisión de sonido.	2 Pérdida de la transmisión de sonido.
20	8000	3 Pérdida de sonido. Los retrasos comienzan a ser notables con dos jugadores.	3 Pérdida de sonido. Los retrasos comienzan a ser notables con dos jugadores.
10	44100	3 Pérdida de la transmisión de sonido.	4 Pérdida del sonido, aunque los retrasos comienzan a notarse con dos jugadores.
10	32000	4 Pérdida de sonido. Los retrasos comienzan a ser notables con tres jugadores.	4 Pérdida del sonido, aunque los retrasos comienzan a notarse con dos jugadores.
10	16000	4 Pérdida de sonido. Los retrasos comienzan a ser notables con tres jugadores.	4 Pérdida del sonido, aunque los retrasos comienzan a notarse con tres jugadores.

Imágenes transmitidas por segundo	Muestras transmitidas cada segundo	Número de jugadores	
		Versión en UNet	Versión en MLAPI
10	8000	4 Pérdida de sonido. Los retrasos comienzan a ser notables con tres jugadores.	4 Pérdida del sonido con cuatro jugadores, sin retrasos excesivos para un número menor.
5	44100	5 Pérdida de la transmisión de sonido.	5 Pérdida de sonido y altos retrasos en la sincronización. Los retrasos comienzan a ser notables con cuatro jugadores.
5	32000	5 Pérdida de la sincronización de la imagen.	5 Pérdida de sonido y altos retrasos en la sincronización. Los retrasos comienzan a ser notables con cuatro jugadores.
5	16000	6 Pérdida de sonido y altos retrasos en la sincronización. Los retrasos comienzan a ser notables con cinco jugadores.	6 Pérdida de sonido y altos retrasos en la sincronización.
5	8000	6 Pérdida de sonido y altos retrasos en la sincronización. Los retrasos comienzan a ser notables con cinco jugadores.	6 Pérdida de sonido y altos retrasos en la sincronización.

Tabla 4: resultados de la evaluación de los efectos de la comunicación audiovisual sobre el número máximo de jugadores simultáneos.

Para poder tener una mejor comprensión de estos datos, decidimos llevar a cabo un experimento para determinar cuántos jugadores pueden conectarse de forma

simultánea en ausencia de transmisión de vídeo o audio para ambas versiones. El resultado nos permitió observar que hasta 12 instancias ejecutándose en la misma máquina podían mantener una conexión con unos retrasos aceptables para el caso de MLAPI. Con respecto a la versión de UNet, fue posible conectar hasta 10 instancias corriendo en la misma máquina, aunque los retrasos a partir de ocho instancias empezaban a ser notables.

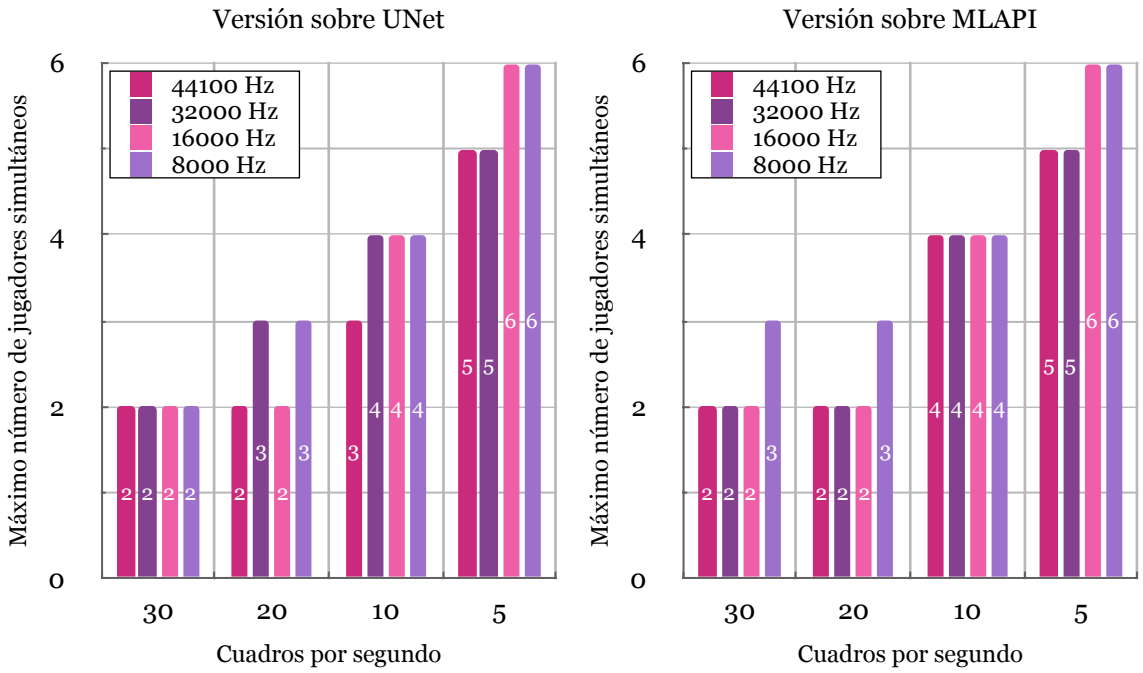

Figura 43: representación gráfica de los resultados de la evaluación de los efectos de la comunicación audiovisual sobre el número máximo de jugadores simultáneos.

A continuación, expusimos las dos versiones en un entorno de uso menos ideal, procediendo a la ejecución de dos instancias situadas en máquinas diferentes conectadas a través de una red wifi local. Es estos tests, se ha observado que ambas versiones mantienen una buena sincronización aún cuando se encuentran activos la cámara y el micrófono a 5 imágenes y 8000 muestras de audio por segundo. No obstante, al incrementar estos valores, el aumento de los retrasos puede provocar desconexiones causadas por errores de tiempo muerto, como es el caso por ejemplo para una transmisión de 20 imágenes y 44100 muestras de sonido por segundo.

Finalmente, cabe destacar que las dos versiones no son compatibles. Al iniciar una instancia de la versión de UNet y una instancia con la versión sobre MLAPI, estas instancias pueden identificarse a través del sistema de descubrimiento, ya que este componente es compatible. No obstante, el proceso de conexión entre un host UNet y un cliente MLAPI, o viceversa, no es posible, produciéndose una desconexión del cliente.

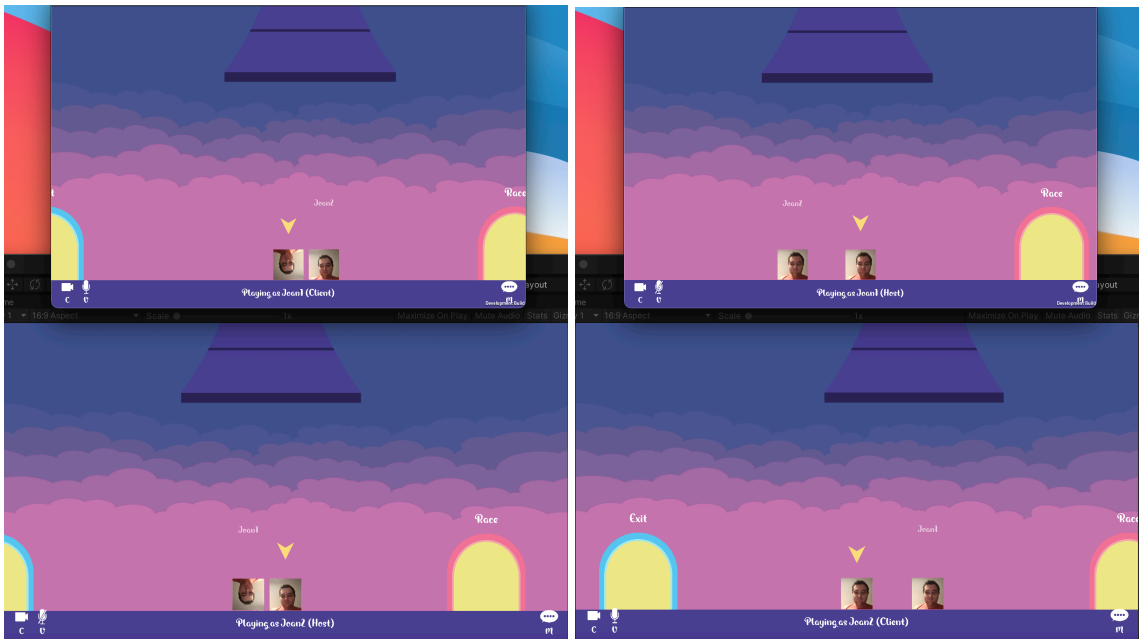

Figura 44: comunicación audiovisual entre dos instancias con UNet, a la izquierda, y entre dos instancias con MLAPI, a la derecha.

7.Conclusiones

A lo largo del desarrollo del proyecto, hemos conseguido definir cuáles son los pasos para la conversión de un juego local en un juego multijugador. Estos pasos nos permiten no solo adaptar proyectos existentes, sino también diseñar nuevos juegos multijugador planteándolos inicialmente como juegos de un solo jugador y luego ayudándonos a determinar sus características y necesidades como multijugador. Los pasos que hemos aprendido y seguido durante nuestro desarrollo se pueden resumir en:

1. Determinar qué objetos deberán presentar un comportamiento distribuido entre todos los dispositivos y qué objetos deberán existir de forma local a cada máquina.

2. Determinar las necesidades de comunicación de cada objeto distribuido, es decir, qué información es necesario compartir entre las instancias.

3. Identificar el comportamiento de cada instancia de un objeto distribuido dependiendo de en qué dispositivo se encuentre.

4. Diseñar el sistema que permita a los jugadores conectarse entre sí.

5. Determinar las necesidades de comunicación entre los jugadores y su soporte. Estas necesidades dependerán del tipo de juego y fomentar una comunicación e interacción adecuada entre los jugadores será fundamental.

A la hora de desarrollar estos pasos, será necesario siempre analizar qué nivel de conectividad implica nuestro juego y cuáles son sus necesidades, así como también la topología más adecuada según nuestro proyecto y presupuesto.

Gracias a nuestro análisis del estado del arte, también hemos podido comprender la situación actual de la solución multijugador de Unity. Sabemos así que, actualmente, el futuro de los juegos multijugador recae sobre MLAPI para los juegos basados en *GameObjets*, y sobre *NetCode* para los juegos basados en DOTS.

Así, a través de nuestro proyecto hemos podido ver las opciones que nos ofrece Unity para el desarrollo de un juego multijugador en tiempo real que cubre todos los aspectos principales de cualquier juego moderno. Hemos podido ver cómo mediante el obsoleto UNet, aún se puede abordar el desarrollo de este tipo de juegos, siendo posible abordar todos los aspectos fundamentales desde el juego en tiempo real hasta la implementación de tablas de clasificación locales, pasando por la comunicación entre los jugadores. No obstante, observamos que la administración de todas estas tareas de comunicación audiovisual junto con la lógica de juego en tiempo real comienzan a estar en los límites de las capacidades y el ámbito de uso objetivo de UNet y MLAPI, aunque para un número reducido de jugadores permiten obtener una comunicación suficiente. Por otro lado, la exploración de MLAPI nos ha permitido ver cómo tiene por objetivo proporcionar todas las funcionalidades necesarias que ya proporcionaba UNet y que, aún estando a día de hoy en desarrollo, se convierte en una buena opción para la creación de juegos multijugador en la actualidad, presentado un futuro prometedor. Consideramos por tanto, tras nuestro desarrollo, que tanto UNet como MLAPI son buenas alternativas para el desarrollo de juegos L1, L2 o L3, cubriendo sus requisitos básicos, aunque su aplicación fuera de un entorno de desarrollo ideal en condiciones de red reales necesita mayor exploración.

Para terminar, cabe destacar por tanto que hemos conseguido alcanzar gracias a nuestro desarrollo e investigación los objetivos que nos propusimos. De esta manera, tenemos el conjunto de pasos necesarios para convertir un juego local en un juego multijugador en red y hemos podido clarificar la situación actual en la que se encuentra la solución multijugador de Unity, elaborando la Figura 2. También hemos explorado las características de la futura solución multijugador de Unity con MLAPI y, finalmente, hemos podido ver mediante los resultados obtenidos cómo las capacidades de Unity permiten cubrir el desarrollo de juegos con restricciones de tiempo real e incluso la comunicación audiovisual entre usuarios, aunque de forma más limitada.

8. Trabajos futuros

Para concluir nuestra memoria, presentamos aquellos tópicos que no han sido cubiertos en nuestro desarrollo pero sí habían sido expuestos en el capítulo 2 y cuya investigación podría ser objetivo de futuras continuaciones de nuestra línea de trabajo. Dichos trabajos podrían ser:

- La investigación del estado de *NetCode* y sus capacidades, tratando de adaptar el proyecto desarrollado a una versión DOTS que hiciera uso de esta API de red.
- El estudio de las capacidades de aplicación de UNet y MLAPI en topologías basadas en servidor dedicado. Esta alternativa implicaría la adaptación del juego actual, el análisis de los requisitos del servidor y el análisis de los costes asociados al mantenimiento para cada una de las versiones.
- La adaptación del proyecto a otras soluciones externas a Unity que hemos presentado en la sección 2, como *Mirror* [24] o *Photon PUN* [34].
- La actualización de la versión actual del proyecto en MLAPI una vez esta termine oficialmente el desarrollo de su versión 1.0.0.

Estas líneas de trabajo permitirían completar el estudio de las alternativas disponibles para el desarrollo de videojuegos multijugador basados en el motor de Unity. Finalmente, como última línea de trabajo a destacar, quedaría por explorar el campo de juegos en red de mundo persistente L4, sus necesidades, y cuáles son las mejores tecnologías para su desarrollo.

9.Referencias bibliográficas

[1] B. House, «Accelerating Unity's new GameObjects multiplayer networking frameworkUnity の新しい GameObject 向けマルチプレイヤーネットワーキングフレームワークの成長を加速させる - Unity Technologies Blog», dic. 03, 2020. https://blogs.unity3d.com/2020/12/03/accelerating-unitys-new-gameobjects-multiplayer-networking-framework/ (accedido may 12, 2021).

[2] Unity Technologies, «Alojamiento de servidores de juegos dedicados de Multiplay | Servicios multijugador de Unity». https://unity.com/es/products/multiplay (accedido may 12, 2021).

[3] E. Juhl, «Announcing UNET - New Unity Multiplayer Technology - Unity Technologies Blog», may 12, 2014. https://blogs.unity3d.com/2014/05/12/announcing-unet-new-unity-multiplayer-technology/ (accedido may 12, 2021).

[4] «Blazing Fast Deterministic Networking Engine | Photon Engine». https://www.photonengine.com/en-US/Quantum (accedido may 14, 2021).

[5] B. House, «Choosing the right netcode for your game 自分のゲームに適したネットコードを選ぼう - Unity Technologies Blog», sep. 08, 2020. https://blogs.unity3d.com/2020/09/08/choosing-the-right-netcode-for-your-game/ (accedido may 12, 2021).

[6] «Cinemachine camera collision», *Unity Forum*. https://forum.unity.com/threads/cinemachine-camera-collision.515626/ (accedido may 18, 2021).

[7] «ClientRpc | Unity Multiplayer Networking». https://docs-multiplayer.unity3d.com//docs/advanced-topics/message-system/clientrpc (accedido jun. 07, 2021).

[8] Unity, *Connected Games: Building real-time multiplayer games with Unity and Google - Unite LA*, (nov. 19, 2018). Accedido: jun. 08, 2021. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=CuQF7hXlVyk&t=8s

[9] M. Wolffelt, R. Hyde y J. Iaccarino. «CSharpMessenger Extended - Unify Community Wiki». http://wiki.unity3d.com/index.php/CSharpMessenger_Extended (accedido may 15, 2021).

[10] «Custom Messages | Unity Multiplayer Networking». https://docs-multiplayer.unity3d.com//docs/advanced-topics/message-system/custom-messages (accedido jun. 07, 2021).

[11] «DarkRift Networking». https://www.darkriftnetworking.com/darkrift2 (accedido may 14, 2021).

[12] Unity Technologies, «DOTS, el nuevo conjunto de tecnología orientada a los datos multiproceso de Unity». https://unity.com/es/dots (accedido may 13, 2021).

[13] B. House, «Evolving multiplayer games beyond UNetUNet を乗り越え、マルチプレイヤーゲームはさらなる進化へ - Unity Technologies Blog». https://

blogs.unity3d.com/2018/08/02/evolving-multiplayer-games-beyond-unet/ (accedido may 12, 2021).

[14] «Fancy Matter Font | Din Studio», *fontspace*, feb. 21, 2021. https://www.fontspace.com/fancy-matter-font-f57302 (accedido jun. 08, 2021).

[15] «Farland Skies - Cloudy Crown | 2D Sky | Unity Asset Store». https://assetstore.unity.com/packages/2d/textures-materials/sky/farland-skies-cloudy-crown-60004 (accedido jun. 08, 2021).

[16] «Getting Started with MLAPI | Unity Multiplayer Networking». https://docs-multiplayer.unity3d.com//docs/getting-started/about-mlapi (accedido may 13, 2021).

[17] «GitHub: Where the world builds software», *GitHub*. https://github.com/ (accedido jun. 10, 2021).

[18] «Install MLAPI | Unity Multiplayer Networking». https://docs-multiplayer.unity3d.com//docs/migration/install (accedido jun. 07, 2021).

[19] «JacksonDunstan.com | Efficiently Keeping Lists Sorted». https://www.jacksondunstan.com/articles/3189 (accedido may 22, 2021).

[20] «Llamada a procedimiento remoto», *Wikipedia, la enciclopedia libre*. jun. 01, 2021. Accedido: jun. 08, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Llamada_a_procedimiento_remoto&oldid=136005649

[21] «Magic Owl Font | shaped fonts», *fontspace*, feb. 23, 2021. https://www.fontspace.com/magic-owl-font-f57364 (accedido jun. 08, 2021).

[22] «Marshalling», *Wikipedia, la enciclopedia libre*. feb. 12, 2021. Accedido: jun. 08, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Marshalling&oldid=133163780

[23] «Migrating From UNet to Unity MLAPI | Unity Multiplayer Networking». https://docs-multiplayer.unity3d.com//docs/migration/migratingtomlapi (accedido jun. 07, 2021).

[24] «Mirror». https://mirror-networking.gitbook.io/docs/ (accedido may 14, 2021).

[25] «MLAPI.NetworkManager | Unity Multiplayer Networking». https://docs-multiplayer.unity3d.com//docs/develop/mlapi-api/MLAPI.NetworkManager (accedido jun. 07, 2021).

[26] B. House, «Multiplayer Connected Games: First steps forward멀티플레이어 커넥티드 게임의 첫걸음マルチプレイヤーコネクティッドゲームへの第一歩 - Unity Technologies Blog», sep. 12, 2018. https://blogs.unity3d.com/2018/09/12/multiplayer-connected-games-first-steps-forward/ (accedido may 12, 2021).

[27] Unity Technologies, «Multiplayer Networking». https://resources.unity.com/unity-engine-roadmap/multiplayer?_ga=2.158997119.473350320.1620921379-1197050217.1620289293 (accedido may 14, 2021).

[28] B. House, «Navigating Unity's multiplayer Netcode transitionUnity のマルチプレイヤー Netcode の移行ガイド - Unity Technologies Blog», jun. 13, 2019. https://blogs.unity3d.com/2019/06/13/navigating-unitys-multiplayer-netcode-transition/ (accedido may 12, 2021).

[29] Unity Technologies, «NetCode | Unity». https://unity.com/es/products/netcode (accedido may 14, 2021).

[30] «Networked scene objects · Issue #207 · Unity-Technologies/com.unity.multiplayer.mlapi», *GitHub*. https://github.com/Unity-Technologies/com.unity.multiplayer.mlapi/issues/207 (accedido jun. 07, 2021).

[31] «NetworkObject binding failed on Host and Client · Issue #328 · Unity-Technologies/com.unity.multiplayer.mlapi», *GitHub*. https://github.com/Unity-Technologies/com.unity.multiplayer.mlapi/issues/328 (accedido jun. 07, 2021).

[32] «NetworkTransform doesn't sync data with new connected client. · Issue #650 · Unity-Technologies/com.unity.multiplayer.mlapi», *GitHub*. https://github.com/Unity-Technologies/com.unity.multiplayer.mlapi/issues/650 (accedido jun. 07, 2021).

[33] «Peer-to-peer», *Wikipedia, la enciclopedia libre*. abr. 10, 2021. Accedido: jun. 08, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=134667623

[34] «Photon Unity 3D Networking Framework SDKs and Game Backend | Photon Engine». https://www.photonengine.com/en-US/PUN (accedido may 14, 2021).

[35] «Prevent key input when InputField has focus», *Unity Forum*. https://forum.unity.com/threads/prevent-key-input-when-inputfield-has-focus.651055/ (accedido may 27, 2021).

[36] «Scrum (desarrollo de software)», *Wikipedia, la enciclopedia libre*. may 28, 2021. Accedido: jun. 21, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Scrum_(desarrollo_de_software)&oldid=135899146

[37] «Stream video through network», *Unity Forum*. https://forum.unity.com/threads/stream-video-through-network.464693/ (accedido may 22, 2021).

[38] «Teorema CAP», *Wikipedia, la enciclopedia libre*. abr. 24, 2020. Accedido: jun. 08, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Teorema_CAP&oldid=125463450

[39] «TextureScale - Unify Community Wiki». http://wiki.unity3d.com/index.php/TextureScale#TextureScale.cs (accedido may 22, 2021).

[40] R. Hauwert y A. Bowell, «The road to 2021», ago. 13, 2020. https://blogs.unity3d.com/2020/08/13/the-road-to-2021/ (accedido may 12, 2021).

[41] «UNET: How do I properly handle Client Authority with interactable Objects? - Unity Answers». https://answers.unity.com/questions/1245341/unet-how-do-i-properly-handle-client-authority-wit.html?childToView=1272413#answer-1272413 (accedido may 17, 2021).

[42] Universitat Politècnica de València, «Unit 8. Communication», en *Concurrency and Distributed Systems*, 2018.

[43] Unity, *Unite Berlin 2018 - The Future of Connected Games Unity and Google Cloud*, (2018). Accedido: jun. 08, 2021. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=UExWjX-S8Jw

[44] Unity Technologies, «Unity - Manual: Multiplayer Overview». https://docs.unity3d.com/Manual/UNetOverview.html (accedido may 12, 2021).

[45] Unity Technologies, «Unity - Manual: Network Authority». https://docs.unity3d.com/Manual/UNetAuthority.html (accedido may 17, 2021).

[46] Unity Technologies, «Unity - Manual: Network Discovery». https://docs.unity3d.com/Manual/UNetDiscovery.html (accedido may 18, 2021).

[47] Unity Technologies, «Unity - Manual: Network Messages». https://docs.unity3d.com/Manual/UNetMessages.html (accedido may 25, 2021).

[48] Unity Technologies, «Unity - Manual: Remote Actions». https://docs.unity3d.com/Manual/UNetActions.html (accedido may 25, 2021).

[49] Unity Technologies, «Unity - Manual: Spawning GameObjects». https://docs.unity3d.com/Manual/UNetSpawning.html (accedido may 18, 2021).

[50] Unity Technologies, «Unity - Scripting API: Microphone». https://docs.unity3d.com/ScriptReference/Microphone.html (accedido may 26, 2021).

[51] Unity Technologies, «Unity - Scripting API: Networking.NetworkBehaviour.OnStartClient». https://docs.unity3d.com/2018.2/Documentation/ScriptReference/Networking.NetworkBehaviour.OnStartClient.html (accedido may 17, 2021).

[52] «Unity - Scripting API: WebCamTexture». https://docs.unity3d.com/ScriptReference/WebCamTexture.html (accedido may 25, 2021).

[53] «Unity (motor de videojuego)», *Wikipedia, la enciclopedia libre*. may 26, 2021. Accedido: jun. 10, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Unity_(motor_de_videojuego)&oldid=135842894

[54] Unity Technologies, «Unity IssueTracker - NetworkBehaviour.OnStartClient is not called for non-player objects when a player reconnects». https://issuetracker.unity3d.com/issues/networkbehaviour-dot-onstartclient-is-not-called-for-non-player-objects-when-a-player-reconnects (accedido may 16, 2021).

[55] «Unity Make MLAPI Official Networking Library for GameObjects – GameFromScratch.com». https://gamefromscratch.com/unity-make-mlapi-official-networking-library-for-gameobjects/ (accedido may 12, 2021).

[56] «Unity NetCode | Unity NetCode | 0.6.0-preview.7». https://docs.unity3d.com/Packages/com.unity.netcode@0.6/manual/index.html (accedido may 12, 2021).

[57] U. Technologies, «Unity Real-Time Development Platform | 3D, 2D VR & AR Engine». https://unity.com/ (accedido jun. 10, 2021).

[58] Unity Technologies, «Unity Technologies Acquires Game Hosting Division of Multiplay from GAME Digital, PLC | Unity». https://unity.com/our-company/newsroom/unity-technologies-acquires-game-hosting-division-multiplay-game-digital-plc (accedido may 12, 2021).

[59] *Unity-Technologies/com.unity.multiplayer.mlapi*. Unity Technologies, 2021. Accedido: may 14, 2021. [En línea]. Disponible en: https://github.com/Unity-Technologies/com.unity.multiplayer.mlapi

[60] *Unity-Technologies/com.unity.multiplayer.mlapi/v.0.1.0*. Unity Technologies, 2021. Accedido: jun 07, 2021. [En línea]. Disponible en: https://github.com/Unity-Technologies/com.unity.multiplayer.mlapi.git?path=/com.unity.multiplayer.mlapi#release/0.1.0

[61] «vis2k / HLAPI Community Edition - 78d159d». https://bitbucket.org/vis2k/hlapi-community-edition/commits/78d159d7da0c71838aec67f9efee9641570b6105 (accedido may 16, 2021).

[62] «What is Scrum?», *Scrum.org*. https://www.scrum.org/resources/what-is-scrum (accedido jun. 21, 2021).