Juegos multijugador en red

PID_00270023

Carles Pairot Gavaldà
Rubén Mondéjar Andreu

Tiempo mínimo de dedicación recomendado: 6 horas

Universitat Oberta de Catalunya

Carles Pairot Gavaldà **Rubén Mondéjar Andreu**

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Javier Cánovas Izquierdo (2020)

Primera edición: febrero 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

En este módulo se presentan los conceptos de redes de computadores que necesitaremos para comprender cómo funcionan las herramientas para el desarrollo de videojuegos multijugador, mediante un breve repaso a las nociones sobre redes, tanto locales como globales, así como una comparativa de las opciones disponibles.

Empezaremos por analizar resumidamente la tecnología que hay tras Internet para saber cómo se realiza la transferencia de información entre dos dispositivos conectados a la red. Posteriormente, se explican los mecanismos que se construyen encima de dicha tecnología para abstraer a los desarrolladores de las problemáticas más habituales.

A continuación, hablaremos de cómo se organizan los dispositivos en red mediante una topología para trabajar de forma conjunta. Analizaremos brevemente las dos vertientes más utilizadas y las ventajas e inconvenientes de cada una de ellas. Además, debemos conocer los problemas más habituales a los cuales nos tendremos que enfrentar una vez nuestro juego empiece a funcionar en una red global. Existen diferentes técnicas para evitar estos problemas y veremos algunas de ellas.

Por último, entraremos de lleno en el desarrollo de videojuegos multijugador con Unity, viendo qué APIs tenemos disponibles y en qué consiste cada una de ellas. Dada la forma en que están construidas y se complementan entre ellas, la recomendación es comenzar nuestro desarrollo utilizando la capa de alto nivel.

Objetivos

En este módulo didáctico se presentan al estudiante los conocimientos necesarios para conseguir los objetivos siguientes:

1. Entender las nociones generales y subyacentes para trabajar en entornos de redes.

2. Comprender las diferentes topologías de red, así como sus características principales.

3. Conocer los conceptos básicos, problemas y soluciones al desarrollar juegos en red.

4. Ser capaz de programar un videojuego multijugador en red con Unity.

1. Breve repaso a las redes e Internet

Si bien el concepto de juego multijugador no se encuentra vinculado exclusivamente a los juegos en línea, es evidente que este es el sistema más usado actualmente para llevar a cabo este tipo de mecánicas. A continuación, se ofrece una breve visión general del conjunto de protocolos TCP/IP, y de los protocolos y estándares asociados que participan en la comunicación por Internet, incluyendo un breve repaso a los que son más relevantes para la programación en red.

1.1. Arquitectura de Internet

La necesidad de comunicar varios dispositivos electrónicos nació paralelamente a estos. Proyectos como ARPAnet crearon las primeras redes de ordenadores a pequeña escala, pero el gran salto cualitativo se produjo con la aparición de Internet como una red de comunicación a escala global. El éxito de Internet reside en que proporciona una infraestructura que permite que un gran número de dispositivos completamente diferentes puedan interactuar entre sí e intercambiar servicios e información.

La comunicación entre dos dispositivos conectados mediante una red se realiza mediante lo que conocemos como protocolo de comunicación, es decir, un conjunto de normas que definen la lengua en la que hablan dos ordenadores entre sí. El conjunto de protocolos más extendido en la actualidad y el que se utiliza actualmente en Internet es conocido como modelo TCP/IP (*Transmission Control Protocol / Internet Protocol*).

1.2. Capas TCP/IP

El modelo TCP/IP está basado en una arquitectura de cinco capas que realizan funciones específicas dentro del proceso de comunicación entre dos aplicaciones, que son:

- **Capa física**. Se trata de la capa de más bajo nivel, donde se codifica la información por medio de señales eléctricas u ondas de radio y se transmite entre dos máquinas por un medio físico.

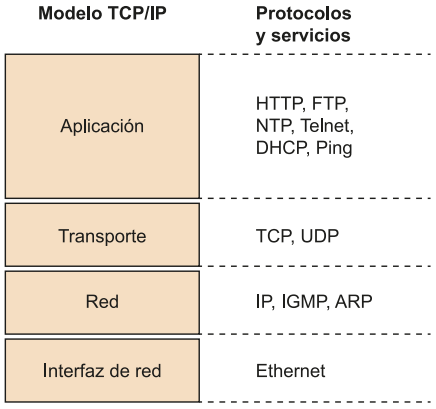
- **Capa de acceso al medio**. Que cubre el hardware y el protocolo de transmisión. Cada implementación usa un estándar de la industria. Por ejemplo: ethernet, RDSI, coaxial, *wireless*, etc.

- **Capa de red**. Responsable del modo en que se comunican varios puntos de una red. Se utiliza un protocolo sencillo de transmisión de paquetes llamado *Internet Protocol* o IP.

- **Capa de transporte**. Responsable del modo en el que se divide y reordena un flujo continuo de información en paquetes IP. También es responsable de asegurar la ausencia de errores. Hay dos protocolos en esta capa: *Transport Control Protocol* (TCP) y *User Datagram Protocol* (UDP). Los analizaremos posteriormente con mayor detalle.

- **Capa de aplicación**. En esta capa se definen los protocolos de las aplicaciones, por ejemplo: SMTP, POP3, IMAP para el correo electrónico, HTTP para contenidos web, etc. Para una aplicación propia es necesario diseñar nuestro propio protocolo de comunicación.

Figura 1. Protocolos y servicios en el modelo TCP/IP



Vamos a analizar cómo se realiza una transmisión de información entre dos ordenadores mediante la arquitectura TCP/IP:

- Cuando una aplicación quiere enviar datos o información a otra, se divide la información en paquetes de datos. El tamaño de estos paquetes viene prefijado por el sistema operativo. Normalmente son bastante pequeños (unos 1.500 bytes por paquete).

- Todos los paquetes tienen que atravesar todas las capas de la arquitectura TCP/IP dentro del ordenador. Cada una de estas capas añade información extra (**encapsulación**) para que se pueda interpretar a qué ordenador va dirigido, y, una vez en la capa física, se realiza la comunicación real con el otro ordenador. Esta comunicación puede pasar por nodos intermedios.

- Una vez que el destinatario recibe el paquete, este atraviesa en sentido inverso la arquitectura usando la información contenida en los datos adicionales hasta que se entrega a la aplicación correspondiente.

Encapsulación

Es un método de diseño modular de protocolos de comunicación en el cual las funciones lógicas de una red son abstraídas ocultando información a las capas de nivel superior.

- El destinatario tiene que recoger todos los paquetes enviados y ensamblarlos de nuevo para poder recuperar la información original.

1.2.1. Direccionamiento

Hemos visto que en cada una de las capas se añade información extra que nos ayuda a controlar la transmisión de los paquetes. Uno de los elementos más importantes que se añade en todos los niveles es la dirección, que permite identificar a qué aplicación y a qué máquina queremos enviar la información. En el modelo TCP/IP existe una dirección para cada nivel:

- **Capa de acceso al medio**. La dirección viene identificada por la MAC (*Media Access Control*) de la tarjeta de red que interviene en la comunicación. Se utiliza para saber a qué máquina de nuestra red local debemos enviar los datos para que se acerquen a su destino.

- **Capa de red**. En esta capa se añade la dirección IP, que identifica la localización exacta del destino. La dirección IP consiste en un número de 32 bits, que normalmente se representa mediante cuatro números decimales de 8 bits separados por puntos.

- **Capa de transporte**. La dirección nos permite saber a qué aplicación se tiene que entregar la información. Esta dirección se representa con un número de 16 bits que se llama *puerto*.

- **Capa de aplicación**. No hay ningún estándar y es la propia aplicación la que guarda y mantiene a qué componente se tienen que distribuir los datos.

1.2.2. Tipos de comunicación

El protocolo de comunicación puede ser orientado a la conexión, en el caso de que se requiera la existencia de un canal persistente entre las dos instancias por las que transmitimos la información, o no orientado a la conexión si no se requiere este canal para transferir los datos.

La comunicación **orientada a la conexión** requiere el establecimiento de un canal de datos mediante una conexión inicial entre las dos instancias que intervienen en la comunicación. Al principio, las dos instancias negocian entre ellas y crean una conexión dedicada para poder hablar entre sí. Una vez se establece el canal, las dos instancias envían y reciben los paquetes de datos a través de este. En la arquitectura TCP/IP, este tipo de conexión se realiza mediante el uso del protocolo TCP. Algunos ejemplos de protocolos a nivel de aplicación que trabajan con él son HTTP, POP3 o FTP.

En cambio, la comunicación **no orientada a la conexión** permite enviar paquetes entre dos instancias sin establecer ningún tipo de conexión inicial. En este caso, el protocolo que se utiliza es el UDP y como ejemplos de aplicaciones basadas en este tipo de conexión tenemos varios sistemas de transferencia de vídeo o audio, DHCP, DNS, etc.

Comparemos los dos protocolos con un poco más de detalle.

Tabla 1. Protocolos TCP / UDP

TCP	UDP
Necesita establecer un canal de comunicación.	No necesita ningún tipo de conexión previa para poder enviar paquetes.
Garantiza que los datos se han transferido íntegramente entre las dos instancias.	No garantiza que los datos se reciban.
Los paquetes se reciben en el mismo orden en que son enviados.	Tampoco garantiza que se reciban en el mismo orden en el que se han enviado.
Los paquetes con los datos pueden tener un tamaño de bytes variable.	Los paquetes (en este caso se llaman datagramas) de datos tienen todos la misma longitud.
Si hay algún problema en la transmisión, los paquetes se reenvían, creando más tráfico en la red.	No existe el reenvío de paquetes.
Es un protocolo lento, ya que tiene que garantizar que todos los datos se reciben y que están en orden.	Es un protocolo rápido, ya que no requiere comprobaciones adicionales.

En el contexto de los videojuegos, la elección del tipo de comunicación es muy importante. Dicha decisión depende de qué tipo de juego queramos implementar.

Por ejemplo, en un juego de estrategia donde cada movimiento es clave para el funcionamiento del sistema, es preferible una conexión TCP que nos garantice que todas las acciones que tome el usuario se ejecuten. En cambio, por ejemplo, en un *First Person Shooter* (FPS), en el que se puede actualizar el sistema multitud de veces por segundo, no podemos utilizar un protocolo TCP porque nos podría ralentizar el sistema, así que posiblemente la mejor opción es utilizar UDP y simplemente no preocuparnos por aquellos paquetes que se pierden.

Otra solución que se nos podría ocurrir sería la de utilizar una solución mixta, es decir, podríamos tener a la vez un canal de control por donde se transmita la información clave con una conexión TCP, y un canal de datos donde se transmita el resto de la información no crucial mediante el protocolo UDP.

Un ejemplo de esta idea sería enviar por TCP los datos referentes a aquellos otros usuarios que se encuentran cerca y con los que podemos llegar a interactuar, y por otro lado, enviar por UDP los datos de usuarios lejanos, con los que la posibilidad de interactuar sea casi nula.

A pesar de esto, **hay que evitar la tentación** de este tipo de aproximación híbrida debido a un problema colateral que esta solución conlleva. El hecho de que ambos protocolos descansen sobre el mismo protocolo IP puede provocar problemas, como congestión y latencia, que indirectamente aporta el protocolo TCP al tráfico de paquetes UDP al utilizarse al mismo tiempo.

Por eso, una alternativa a TCP muy utilizada en el mundo de los videojuegos en red es el R-UDP (*Reliable* UDP). Este protocolo es muy similar a UDP, pero mediante una implementación ligera consigue que tenga la misma fiabilidad que TCP, pero sin el *overhead* extra y los problemas mencionados.

UDP en Unity

Unity proporciona y utiliza diferentes modalidades de UDP, incluyendo R-UDP.

2. Programación en red

Control de los datos

La programación se puede efectuar de distintas maneras, según el control que queramos tener sobre lo que se envía y se recibe.

La programación de un juego en red requiere dos elementos imprescindibles. En primer lugar, necesitamos crear las conexiones entre los clientes, así como saber cómo enviar y gestionar la información. Y, en segundo lugar, necesitamos diseñar o incorporar técnicas que nos permitan abstraernos de la comunicación entre las diferentes instancias remotas de un mismo juego. Es decir, tenemos que diseñar o utilizar diferentes mecanismos que nos permitan enviar y replicar el estado de nuestro juego y de los elementos que lo componen.

La primera opción es utilizar las herramientas de bajo nivel como los llamados *sockets*, los cuales nos proporcionan un acceso de directo a la capa de red. En este sentido, tenemos un control total de cada byte de información que se transmite por la red, pero esto también implica que tenemos que encargarnos de cualquier problema que pueda suceder, así como tratar con múltiples conexiones que requieren estructuras más complejas.

Otra opción es utilizar un mecanismo de **alto nivel**, que gestione mediante abstracciones todas las conexiones concurrentes y el estado de las mismas. En este caso, tenemos menos control, pero si esta capa superior está implementada de forma eficiente, vamos a reducir mucho el trabajo necesario para poder implementar las conexiones, y nos podremos centrar en el diseño del juego.

Ambas opciones son perfectamente válidas y complementarias a la hora de implementar un videojuego.

Siguiendo este orden, primero veremos la programación básica en red y a continuación hablaremos de los mecanismos que podemos aplicar según los requerimientos que tengamos.

2.1. Sockets

La herramienta básica para programar en red se basa en el concepto de *socket*.

Un *socket* es una entidad que permite que un ordenador intercambie datos con otros ordenadores. Un *socket* identifica una conexión particular entre dos ordenadores y se compone siempre de cinco elementos: una dirección de red local (identificador IP), un puerto de la máquina local, una dirección de red destino, un puerto de la máquina destino y el tipo de protocolo (TCP o UDP).

El concepto de *socket* se desarrolló en la Universidad de Berkeley cuando empezaron a comunicar varios ordenadores UNIX. A partir de esa experiencia, se desarrolló una biblioteca sencilla que permitía aprovechar este trabajo y escribir nuevos programas capaces de comunicarse entre ellos.

2.1.1. Uso de *sockets* TCP

Dado que TCP es un protocolo que requiere conexión, primero es necesario establecerla entre los dos ordenadores antes de poder transmitir información. Además, TCP debe mantener el estado para poder reenviar los paquetes perdidos, y este estado tiene que ser guardado de alguna forma. Siguiendo la API de Berkeley, son los propios *sockets* los que guardan el estado de la conexión. Esto significa que un *host* necesita un *socket* único para cada conexión TCP que mantenga abierta.

En nuestro caso, hablaremos de cómo implementar los ejemplos básicos mediante la programación en C#, que podemos incluir aplicando una adaptación simple dentro de nuestros *scripts* en Unity. En cualquier caso, los pasos básicos para establecer una comunicación mediante TCP serían:

Sockets en Unity

La biblioteca que nos permite usar *sockets* en Unity es *System.Net.Sockets*.

- En primer lugar, tenemos que importar las bibliotecas de *sockets* para poder tener acceso a la interfaz de red.

- Una vez tenemos acceso a la red, abrimos un nuevo *socket* por el que vamos a enviar y recibir los datos. En este ejemplo empezamos por abrir un *socket* TCP, en la dirección y puerto del servidor, y de esta forma obtenemos el canal de *streaming*.

- Cuando tenemos el canal abierto, podemos enviar y recibir datos por él. En este caso, es un ejemplo de cómo enviar un texto de prueba, pero como veremos, será el protocolo de nivel aplicación (en este ejemplo, HTTP) el que determina la forma en que se va a intercambiar la información.

A modo de ejemplo y para ilustrar mejor cómo funciona este protocolo, vamos a ver qué código necesitamos para poder realizar un ejemplo sencillo. Primero veremos cómo instanciar y hacer funcionar el servidor, quien recibe las conexiones y envía la información, y a continuación veremos el cliente, que se encarga de conectar y recibir esta información.

Por lo tanto, empezaremos con la clase ***TcpClient***, la cual permite tanto recibir conexiones durante el proceso de escucha, como a la inversa, solicitar los datos de un recurso de Internet mediante TCP. Las funciones y las propiedades de *TcpClient* resumen los detalles para gestionar el *socket* y también para solicitar y recibir datos mediante TCP.

Para esperar una conexión TCP, se requiere establecer un puerto TCP que el cliente debe conocer. *Internet Assigned Numbers Authority* (IANA) define los números de puerto para los servicios comunes como SSL o FTP. Los servicios fuera de la lista IANA pueden ser puertos dentro del intervalo de los números 1.024 a 65.535. En la imagen siguiente se muestra cómo crear un servidor horario de red mediante la clase ***TcpListener*** escuchando en el puerto 1755. Habitualmente deberíamos utilizar el puerto 13, pero al ser inferior a 1.024, Unity nos va a denegar el acceso al tratarse de un puerto reservado. En este ejemplo, cuando se acepta una solicitud de conexión entrante, el servidor horario responde con la fecha y hora actual de la máquina donde se ejecuta:

```
using UnityEngine;
using System;
using System.Net.Sockets;
using System.Text;
using System.Net;
using System.Threading;

public class TCPTimeServer : MonoBehaviour {
    private Thread socketThread;
    private TcpListener listener;
    private bool done = false;
    private readonly int port = 1755;

    void Start() {
        Application.runInBackground = true;
        StartServer();
    }

    void StartServer() {
        socketThread = new Thread (NetworkCode);
        socketThread.IsBackground = true;
        socketThread.Start();
    }

    private void NetworkCode() {
        listener = new TcpListener (IPAddress.Any, port);
        listener.Start();

        while (!done) {
            Debug.Log ("Waiting for connection...");
            TcpClient client = listener.AcceptTcpClient();

            Debug.Log ("Connection accepted.");
            NetworkStream ns = client.GetStream();

            byte[] byteTime = Encoding.ASCII.GetBytes (DateTime.Now.ToString());
```

```
            try {
                ns.Write(byteTime, 0, byteTime.Length);
                ns.Close();
                client.Close();
            }
            catch (Exception e) {
                Debug.Log (e.ToString());
            }
        }
    }

    void StopServer() {
        done = true;

        if (socketThread != null) {
            socketThread.Abort();
        }

        if (listener != null)
        {
            listener.Stop();
            Debug.Log ("Server disconnected!");
        }
    }

    void OnDisable() {
        StopServer();
    }
}
```

Como se puede observar, *TcpListener* acepta las solicitudes entrantes con la función *AcceptTcpClient()* y para cada una de ellas crea un *socket* (*TcpClient*) que administra la conexión al cliente. Una particularidad en este código es que se ha tenido que crear un hilo de ejecución (*Thread*) separado para gestionar de forma concurrente las distintas conexiones que este servidor puede recibir al mismo tiempo. Si no se hiciera de esta forma, el proceso se quedaría colgado hasta recibir una conexión nueva. Es por ello que la función *Start()* crea un nuevo *Thread* que ejecuta en paralelo la función *NetworkCode()*. Análogamente, la función *OnDisable()* realiza la finalización limpia del proceso abortando el *Thread* en curso y liberando el puerto, ya que mientras este está siendo utilizado, ningún otro servidor podrá utilizarlo en la máquina donde se ejecuta dicho proceso.

Probando el código

Para probar estos códigos de ejemplo, basta con añadir un *script* en un *GameObject* vacío y ejecutar el juego.

Por otro lado, para que un cliente pueda establecer una conexión TCP, además del puerto donde se está ofreciendo el servicio, requiere conocer también la dirección del dispositivo de red que hospeda el servicio. En el siguiente ejem-

plo se muestra la configuración de *TcpClient* para conectarse a un servidor en el puerto TCP 1755 y leer de este la información de tiempo enviada por el servidor.

```
using UnityEngine;
using System;
using System.Net.Sockets;
using System.Text;

public class TCPTimeClient : MonoBehaviour {
    private String  host = "127.0.0.1";
    private int     port = 1755;

    void Start() {
        try {
            TcpClient client = new TcpClient (host, port);

            NetworkStream ns = client.GetStream();

            byte[] bytes = new byte[1024];
            int bytesRead = ns.Read (bytes, 0, bytes.Length);

            Debug.Log (Encoding.ASCII.GetString (bytes, 0, bytesRead));

            client.Close();

        } catch (Exception e) {
            Debug.Log(e.ToString());
        }
    }
}
```

2.1.2. Uso de sockets UDP

A continuación, siguiendo el mismo esquema que en el caso de TCP, se expone cómo trabajar a nivel cliente-servidor con el protocolo UDP.

Recordemos que el **protocolo de datagramas de usuario** (UDP) es un protocolo simple que ofrece un mayor desempeño para entregar datos a un *host* remoto. Sin embargo, el protocolo UDP no es orientado a la conexión y por tanto no garantiza el envío de los datagramas UDP al extremo remoto, ni que su recepción sea en el orden en el cual fueron enviados.

A diferencia de TCP, en este protocolo el esquema es más bien productor-consumidor que cliente-servidor, dado que no existe una conexión entre ellos. Para poder enviar un datagrama mediante UDP, el productor debe conocer la dirección de red del dispositivo que quiere consumir esta información, así como el número de puerto UDP que se utiliza para comunicarse.

Además, este protocolo permite el envío de mensajes a múltiples *hosts* al realizar envíos masivos de mensajes a grupos, acción conocida como *multicast* o indiscriminados como *broadcast*, dependiendo del tipo de difusión, red involucrada y versión de protocolo IP. Es fácil imaginar que este tipo de sistema de mensajería es especialmente útil si estamos realizando acciones comunes en múltiples máquinas, como los juegos multijugador en red.

Vamos a ver a modo de ejemplo cómo podríamos implementar envíos de tipo *broadcast*. Las difusiones se pueden dirigir a partes específicas de una red estableciendo todos los bits de identificador del *host*. Por ejemplo, para enviar una difusión para todos los *hosts* de una LAN identificada por 192.168.70.x, se debe utilizar la dirección 192.168.70.255. Para utilizar esta modalidad se requiere utilizar direcciones de red especiales en redes basadas en IPv4 mediante una máscara de red concreta (255.255.255.0).

Ved también

Para más información sobre la API de C# y *sockets* podéis revisar la documentación oficial.

A continuación, veremos el ejemplo de producción de mensajes de tipo *broadcast* donde se abre un *socket* para enviar los datagramas de UDP a la red LAN, utilizando el puerto 11000. En este ejemplo se envía un mensaje de texto específico que se espera al otro lado.

```
using UnityEngine;
using System.Net;
using System.Net.Sockets;
using System.Text;

public class UDPSender : MonoBehaviour {
    private int     port       = 11000;
    public  string  message    = "hello";

    void Start() {
        Socket s = new Socket (AddressFamily.InterNetwork, SocketType.Dgram,
            ProtocolType.Udp) {
            EnableBroadcast = true
        };

        byte[] sendbuf = Encoding.ASCII.GetBytes (message);
        IPEndPoint ep = new IPEndPoint (IPAddress.Broadcast, port);

        s.SendTo (sendbuf, ep);

        Debug.Log ("Message sent to the broadcast address.");
```

```
    }
}
```

Finalmente, veremos el ejemplo complementario de código donde se utiliza la clase *UdpClient* para recibir los datagramas UDP enviados a la dirección de difusión en el puerto 11000. En este caso, todos consumen los datagramas que lleguen a ese puerto en esta LAN (192.168.70.x) recibiendo el mensaje de texto enviado y producido por el ejemplo anterior.

```
using UnityEngine;
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;

public class UDPListener : MonoBehaviour {
    private Thread socketThread;
    private UdpClient listener;
    private bool done = false;
    private readonly int port = 11000;

    void Start() {
        Application.runInBackground = true;
        StartServer();
    }

    void StartServer() {
        socketThread = new Thread (NetworkCode);
        socketThread.IsBackground = true;
        socketThread.Start();
    }

    private void NetworkCode() {
        listener = new UdpClient (port);
        IPEndPoint group = new IPEndPoint (IPAddress.Any, port);

        try {
            while (!done) {
                Debug.Log ("Waiting for broadcast...");
                byte[] bytes = listener.Receive (ref group);

                Debug.LogFormat ("Received broadcast from {0} :\n {1}\n",
                            group.ToString(),
                            Encoding.ASCII.GetString (bytes, 0, bytes.Length));
            }
```

```
            } catch (ThreadAbortException) { }
            catch (Exception e) {
                Debug.Log (e.ToString());
            }
        }

        void StopServer() {
            done = true;

            if (socketThread != null) {
                socketThread.Abort();
            }

            if (listener != null) {
                listener.Close();
                Debug.Log ("Server disconnected!");
            }
        }

        void OnDisable() {
            StopServer();
        }
    }
```

2.1.3. Procesos

Una vez tenemos la lógica para establecer conexiones y poder enviar y recibir paquetes implementada, el siguiente paso es adaptar el código para que gestione las conexiones y la espera de los diversos paquetes a recibir. Para ello existen dos opciones diferentes:

- **De forma asíncrona**. Simplemente se revisa cada cierto tiempo si ha llegado un paquete. Si no ha llegado ninguno, seguimos ejecutando nuestro código y al cabo de un tiempo determinado volvemos a revisar si ha llegado alguno. Esto nos permite no tener que esperar activamente en nuestro código la llegada de nueva información.

- **De forma síncrona**. En este caso cuando miramos si ha llegado un paquete, bloqueamos el sistema y esperamos hasta que llegue el primer paquete. El programa no continúa ejecutándose a no ser que tengamos un sistema multihilo.

Dependiendo de la API que utilicemos para enviar y recibir mensajes, tendremos o no disponible la opción asíncrona. La API básica que se nos ofrece es la síncrona y es por ello por lo que, para trabajar de manera eficiente con *sockets* de forma síncrona, hay que complementar nuestro juego con un sistema multihilo, tal y como hemos visto en los ejemplos anteriores. Y esto es así porque en una comunicación siempre es necesario hacer esperas para recibir información del otro *host* (servidor o cliente), y ese tiempo y recursos son cruciales para otros aspectos de la ejecución, especialmente cuando se trata de videojuegos.

Sistema multihilo

El uso de varios hilos en un videojuego nos permite crear diferentes tareas que están trabajando todo el tiempo. En el caso de ordenadores o consolas con varias CPU, la programación multihilo nos permite sacar el máximo partido a todas ellas.

Un sistema multihilo está muy relacionado con el sistema operativo. Por ejemplo, en Windows debemos trabajar con su propia API de multihilo y en entornos UNIX, con POSIX. En este punto nos interesa trabajar con herramientas que nos abstraigan del sistema operativo para hacer nuestro código más portable, como lo hace Unity por ejemplo.

Otro punto interesante a tener en cuenta es que, en varios tipos de juegos, el usuario demanda hacer uso de otras **funcionalidades paralelas**, como por ejemplo poder hablar por micrófono con los demás jugadores. Para implementar este punto con eficiencia, debemos dedicar muchos recursos y tener un amplio conocimiento de redes y transferencia de sonido (*streaming*) en tiempo real.

Otra opción más recomendable, es delegar este tipo de funcionalidades a un servicio externo a nuestro juego que nos permita integrarnos con él fácilmente sin necesidad de incorporar de forma explícita el mecanismo ni las bibliotecas requeridas en nuestro juego.

2.2. Formatos de transmisión de datos

Como hemos visto en los ejemplos de código anteriores, para transmitir objetos entre instancias de red de un juego multijugador, el juego debe dar formato a los datos de los objetos antes de poder ser enviados por un protocolo de capa de transporte. En este sentido, la serialización de datos de objeto es solo el primer paso en la transmisión de estado entre los *hosts*. Otra herramienta de gran utilidad para minimizar el tamaño de los datos transmitidos y aumentar el rendimiento es la compresión.

En este apartado también comentaremos la técnica de replicación que soporta la sincronización del mundo y del estado de los objetos entre los procesos remotos, y que usualmente se realiza mediante llamadas remotas a procedimientos que permiten abstraer la comunicación para crear código de alto nivel más comprensible y fácil de mantener.

2.2.1. Serialización

El término serialización se refiere al acto de convertir un objeto contenido en memoria hacia una serie lineal de bits. Estos bits se pueden almacenar en el disco o se pueden enviar a través de una red, siendo posteriormente restaurados a su formato original.

Para transmitir objetos entre instancias de red de un juego multijugador, estas deben dar formato a los datos que contienen los objetos a enviar de manera que puedan ser transmitidos por un protocolo de capa de transporte.

El mecanismo de *stream* (flujo) consiste en un flujo de datos; es una fuente de datos, normalmente en bytes o caracteres, utilizado tanto en la escritura como en la lectura de ficheros o también en la comunicación entre conexiones de red.

Un *stream* se usa básicamente para abstraernos de las tareas de serialización explícita de los datos. Estos datos varían desde estructuras básicas como booleanos o enteros, hasta estructuras de datos muy complejas que incluyen referencias a otras estructuras de datos.

En nuestro caso, utilizamos un *stream* de red, que encapsula un *socket*, proporcionando acceso a los métodos para enviar o recibir datos, en sus diferentes variantes dependiendo del tipo de datos que se quieran utilizar.

Para ilustrar este mecanismo, y como hemos visto en el ejemplo de TCP, podemos obtener del *socket* de la conexión un *stream* de red, que nos permite escribir sobre él y en consecuencia enviar por la red los datos de la fecha y hora actuales.

```
NetworkStream ns = client.GetStream();

byte[] byteTime = Encoding.ASCII.GetBytes (DateTime.Now.ToString());

ns.Write(byteTime, 0, byteTime.Length);
```

Notad además que, previamente al envío, se ha codificado esta información en formato ASCII. La codificación es un mecanismo muy relacionado con la serialización de cadenas de caracteres o textos que permite codificar, transformar e interpretar de texto a bits y viceversa.

Encoding.ASCII

Esta clase permite obtener una codificación del texto proporcionado usando el juego de caracteres ASCII.

2.2.2. Compresión

Las primeras conexiones

En los primeros días de los juegos multijugador, los juegos tuvieron que conformarse con 2.400 bytes por segundo o menos, dependiendo del tipo de conexión.

Como hemos visto, con las herramientas de serialización de datos es posible escribir código que nos permita enviar y recibir los objetos del juego a través de la red. Sin embargo, no sería un código eficiente aquel que no respeta las limitaciones del ancho de banda, la cantidad de bits que se pueden gestionar por unidad de tiempo impuestas por la propia red.

Hoy, los desarrolladores de juegos somos más afortunados al disfrutar de conexiones de alta velocidad en muchos órdenes de magnitud, pero todavía tenemos que preocuparnos de cómo utilizar ese ancho de banda de una forma lo más eficiente posible.

Pongamos por ejemplo un juego con un mundo enorme con cientos de objetos en movimiento. En este caso, el envío exhaustivo en tiempo real de los objetos potencialmente accesibles por una gran cantidad de jugadores activos es capaz de saturar la conexión sin importar la capacidad disponible del ancho de banda, ya que este debe ser compartido por todos ellos.

Para reducir esta problemática hay que comenzar en el nivel más bajo, mediante el examen de las técnicas comunes para la compresión de datos a nivel de bits y bytes. Es decir, una vez que un juego ha determinado que una pieza específica de datos ha de ser enviada, permite realizar esta operación utilizando el mínimo de bits posible.

Siguiendo con el ejemplo de código del servidor TCP, antes de enviar los datos por la red, podemos no solo codificarlos, sino que podemos aplicar compresión para reducir el tamaño del paquete que vamos a enviar:

```
            NetworkStream ns = client.GetStream();

            byte[] byteTime = Encoding.ASCII.GetBytes (DateTime.Now.ToString());

            byte[] compress = Compress (byteTime);
            ns.Write (compress, 0, compress.Length);
```

El siguiente ejemplo muestra un algoritmo de ejemplo que hace uso de la conocida biblioteca *gzip* y un *stream* en memoria para comprimir un *array* de bytes:

```
    private byte[] Compress(byte[] raw) {
        using (MemoryStream memory = new MemoryStream()) {
```

```
            using (GZipStream gzip = new GZipStream(memory,
            CompressionMode.Compress, true)) {
                gzip.Write(raw, 0, raw.Length);
            }
            return memory.ToArray();
        }
    }
```

La mayoría del código de serialización de objetos sigue el mismo patrón: se recorre cada variable miembro de la clase de un objeto y se serializa el valor de dicha variable. Puede haber algunas optimizaciones, pero en general el código suele ser el mismo. De hecho, es tan similar entre cada objeto que las herramientas de serialización permiten utilizar un solo método de escritura y otro de lectura para manejar la mayoría de las necesidades de serialización.

2.2.3. Replicación

La replicación de un objeto implica algo más que el envío de sus datos serializados de un *host* a otro. Para tener éxito, un juego de varios jugadores simultáneos debe hacer sentir a los jugadores que están interaccionando en el mismo mundo. Cuando un jugador abre una puerta o acaba con un enemigo, todos los demás necesitan ver esa acción y sus resultados.

Esta experiencia compartida es el reflejo del estado del mundo en cada *host* e implica intercambiar toda la información necesaria para mantener la coherencia entre el estado de cada huésped. En primer lugar, un protocolo de nivel aplicación debe definir todos los posibles tipos de paquetes, y el módulo de red debe etiquetar paquetes que contienen datos de objeto como tal. Cada objeto tiene un identificador único en el juego general (no solo en la partida del jugador que lo ha generado), de modo que el *host* receptor conoce el estado del objeto apropiado. Por último, cada clase de objeto necesita un identificador único para que el *host* receptor pueda crear un objeto de la clase correcta, si no existe ninguno.

En los juegos a pequeña escala es posible crear un mundo compartido entre *hosts* mediante la replicación de cada objeto en el mundo en cada paquete saliente. En cambio, en los juegos más grandes no se pueden ajustar los datos de replicación para todos los objetos en cada paquete, por lo que se debe emplear un protocolo que admita la transmisión de los cambios en el estado del mundo. Cada cambio puede contener acciones de replicación para crear un objeto, actualizar un objeto, o destruir un objeto.

Para una mayor eficacia, las acciones de actualización de objetos pueden enviar los datos de serialización de un subconjunto acotado de las variables del objeto. El uso de este envío parcial depende de la topología de la red global y la fiabilidad del protocolo a nivel de aplicación. En función de la topología de la red del juego, hay varias maneras de crear y hacer cumplir la coherencia entre los estados del mundo entre *hosts* remotos.

Una forma sería mediante la **replicación total**. Este método, bastante común, consiste en tener un servidor para transmitir el estado del mundo a todos los clientes conectados. Los clientes reciben este estado de transmisión y actualizan su propio estado del mundo en local. De esta manera, todos los jugadores experimentan el mismo estado. El estado del mundo se puede definir como el estado de todos los objetos del juego en ese mundo. Por lo tanto, la tarea de transmitir el estado del mundo se puede descomponer en la tarea de transmitir el estado de cada uno de sus objetos.

Replicación total

En la replicación total, cuando el *host* receptor termina de deserializar un objeto, este solo utiliza parte de los datos.

Al utilizar código de replicación para múltiples objetos, podemos plantearnos realizar la réplica del mundo entero, ya que es probable que sea mucho más sencillo de esta forma. En este sentido, si la representación del mundo en un juego es lo suficientemente pequeño, entonces el estado entero del mundo puede caber completamente dentro de un único paquete.

Otra opción, sería la **replicación parcial**, enviando solo los cambios del estado del mundo. Debido a que cada anfitrión mantiene su propia copia, no es necesario replicar todo el estado en un solo paquete. En su lugar, el remitente puede crear paquetes que representan cambios en el estado del mundo, y el receptor puede entonces aplicar estos cambios a su propio estado mundial. De esta manera, un remitente puede usar múltiples paquetes para sincronizar un mundo muy grande con un *host* remoto.

Replicación parcial

La replicación parcial solo funciona si el remitente tiene una representación exacta del estado del mundo actual para determinar los cambios necesarios a replicar.

Cuando se envía una actualización de objeto, el remitente podría no necesitar enviar cada propiedad en el objeto. El remitente puede serializar solo el subconjunto de variables que han cambiado desde la última actualización. Para permitir esto, se puede utilizar un campo de bits para representar las variables serializadas. Cada bit puede representar una variable o grupo de propiedades que se va a serializar. Por eficiencia y escalabilidad, esta última estrategia puede ser muy interesante.

Dado que Internet es inherentemente poco fiable, no se puede suponer que el estado del mundo del receptor se base en los últimos paquetes transmitidos por el emisor. Para este fin, los *hosts* deberían enviar paquetes a través de TCP, que garantizaría la fiabilidad; o alternativamente utilizar un protocolo de nivel de aplicación diseñado en la parte superior de UDP que proporcione fiabilidad, como R-UDP.

Como podemos intuir, la replicación de objetos es un mecanismo clave en los juegos de varios jugadores y será un ingrediente fundamental a la hora de desarrollar un videojuego multijugador en red dependiendo además de las topologías de red en las que se base.

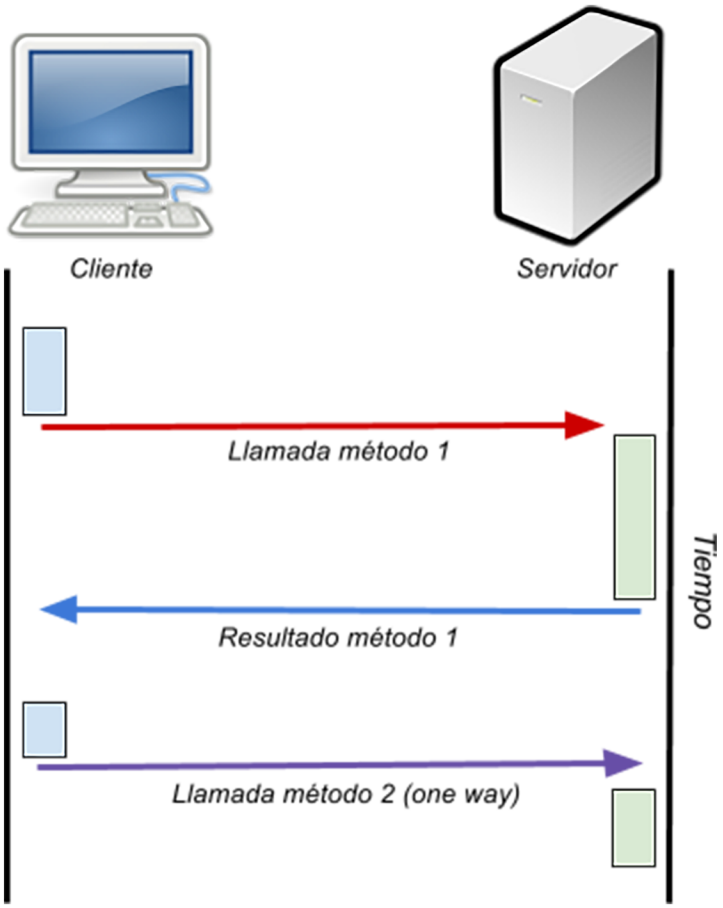
2.2.4. Llamadas remotas

Por último, la **llamada remota a un procedimiento** (RPC, *Remote Procedure Call*) es una herramienta que se utiliza para abstraernos de cómo se realiza la ejecución de un proceso hospedado en otra máquina y se espera su respuesta con el resultado de dicha ejecución.

El mecanismo de RPC proporciona una capa de programación por encima de las librerías de red y evita los problemas más comunes (sincronización, envío de información...). Este es especialmente útil en videojuegos donde se requiere replicar más que el objeto de datos de estado entre las instancias distribuidas del juego.

Esta abstracción es ampliamente utilizada para simplificar los protocolos de comunicación, tanto bidireccionales en llamadas a funciones de los que hay que esperar una respuesta, como en comunicaciones unidireccionales (*one way*) donde no se espera ninguna respuesta de la invocación remota.

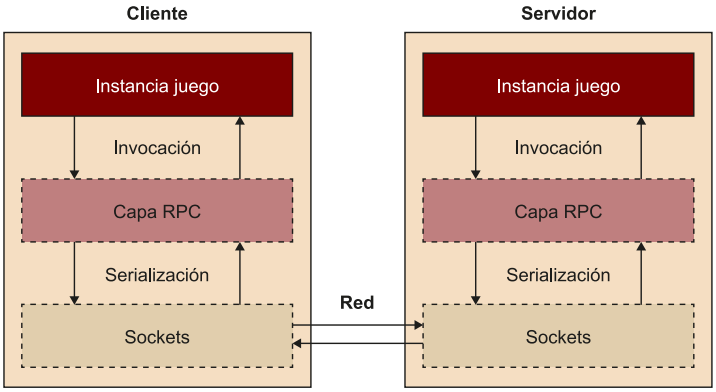
Figura 2. Comunicación bidireccional y unidireccional (*one way*)



Un ejemplo de este último caso, en el que se quiere transmitir información no relacionada con el estado del objeto y sin retorno, podría ser la orden de una acción remota en otro *host* (por ejemplo, producir el sonido de una explosión). Dado que en el *host* remoto ya debe existir un objeto responsable de realizar dicha acción con su *script* asociado, lo que interesa es claramente ejecutar ese código remoto de la forma más simple posible. Esto lo veremos más claro cuando estudiemos el ejemplo de *Tanks!* que trabajaremos más adelante en los retos de este módulo.

Así pues, una llamada a un procedimiento remoto se ejecuta en un *host* origen, que causa una reacción en un procedimiento de otro *host*, siendo cada invocación una llamada única, indicando el identificador del objeto remoto con sus variables asociadas enviadas como parámetros, siendo serializadas en origen y deserializadas en destino.

Figura 3. Llamada a un procedimiento remoto



Recomendación

Al igual que en otras ocasiones se recomienda, si existe la opción, utilizar la capa de RPC que nos proporcione el *framework* utilizado, como es el caso de Unity.

Notad que esta abstracción también nos permite no tener que guardar la información explícita de los *hosts* de los otros jugadores o del servidor, si no que se basa en centrarnos en la información de los objetos remotos que, se ejecuten donde se ejecuten, serán igualmente invocados gracias al sistema subyacente.

Existe tanto la posibilidad de utilizar uno de los diversos protocolos conocidos a nivel de aplicación disponibles (por ejemplo, XML-RPC), como la opción de construir nuestra propia capa de RPC.

3. Tipologías de juego de red

Hay diferentes maneras de conectar los distintos *hosts* o instancias de jugador de nuestro juego. La forma más clásica es separar las conexiones según el rol que tiene cada uno de los *hosts* que intervienen en la comunicación. En el caso de que queramos una jerarquía de comunicación, necesitaremos un sistema cliente-servidor, y en el caso de que queramos que todas instancias se comporten de la misma forma, necesitaremos una arquitectura *peer-to-peer*.

3.1. Cliente-servidor

En este tipo de comunicación, definiremos una instancia como servidor y el resto como clientes:

- El **servidor** es aquel que permanece a la espera de peticiones de clientes y las va sirviendo a medida que van llegando.
- El **cliente** es responsable de realizar peticiones al servidor.

Por ejemplo, cada vez que un cliente quiere enviar un mensaje o alguna información a los otros clientes, esta información la envía a través del servidor, siendo este el que la distribuye entre los clientes destino.

Juegos cliente-servidor

Algunos ejemplos son *World of Warcraft* o *Clash of Clans*.

Estas son algunas ventajas que obtenemos al utilizar un sistema cliente-servidor:

- La seguridad se puede controlar mejor, ya que el servidor puede detectar si alguno de los clientes está realizando alguna acción inapropiada.
- En el caso de que haya datos persistentes, permite garantizar su integridad.
- Normalmente, los servidores son sistemas dedicados con un hardware específico, lo que aumenta su rendimiento frente a ordenadores estándares.
- Al ser un sistema centralizado, es mucho más fácil gestionar los usuarios y las partidas.

Por otro lado, estos son algunos de los inconvenientes asociados al uso de un sistema cliente-servidor:

- Necesitamos crear dos tipos diferentes de lógica, una que se encargue de las tareas propias del servidor, como la gestión del juego; y otra del cliente, que nos permita jugarlo.

- Si queremos utilizar un servidor dedicado hay que considerar el coste elevado del hardware. Además, se precisan administradores de red y técnicos de sistemas que se encarguen de configurar y mantener el servidor actualizado, así como de garantizar su buen funcionamiento.

- En el caso de que haya un problema en el servidor, se interrumpe completamente el desarrollo del juego.

- La conexión del servidor puede suponer un cuello de botella en el rendimiento global del sistema.

Finalmente, para estructurar un juego en red en el que los jugadores están directamente conectados al servidor sin conocer al resto de clientes, nos encontramos con las dos aproximaciones contrapuestas.

La primera aproximación es conocida como ***authoritative server***. Esta aproximación requiere de un servidor para realizar la simulación de todo el mundo, la aplicación de las reglas del juego y el procesamiento de las interacciones provenientes de todos los clientes. Cada cliente envía sus interacciones al servidor y continuamente recibe las actualizaciones de estado, que por sí mismo nunca modifica. Esto permite al servidor escuchar a todos los clientes y posteriormente decidir cómo se actualiza el estado.

Por ello se puede observar una diferencia clara entre lo que el jugador quiere realizar y lo que realmente ocurre. Una ventaja clara respecto a evitar trampas de los clientes, ya que, por ejemplo, los clientes no pueden engañar al servidor sobre el estado del juego (como la falsa muerte de otro jugador), ya que es el servidor quien debe decidirlo.

Una potencial desventaja de esta estrategia es el tiempo que tardan los mensajes en viajar entre tantos nodos, la posible congestión o cuello de botella que esto provoca, o confiar en un punto único de fallo (SPOF, *Single Point of Failure*).

SPOF

Es un componente que, tras un fallo en su funcionamiento, provoca un fallo global en todo el sistema.

En contraposición a esta estrategia, encontramos la conocida justamente como ***non-authoritative server***. En esta ocasión, el servidor no se encarga de controlar el resultado de cada interacción del jugador. Son los propios clientes los que procesan localmente las entradas y la lógica del juego, y luego envían el resultado de cualquier acción determinada al servidor. Acto seguido, el servidor sincroniza todas las acciones del estado del mundo que recibe.

Esta segunda opción hace más simple la implementación del servidor, que queda relegado a tareas de retransmisión de mensajes sin lógica de procesamiento asociada, siendo los clientes los propietarios reales de sus objetos y los únicos autorizados a enviar actualizaciones de esos objetos a través de la red.

Actualmente, muchos de los juegos que utilizan la arquitectura cliente/servidor se basan en una aproximación híbrida de ambas estrategias, donde algunos comportamientos se tratan de forma autoritativa y otros no. Hay que tener en cuenta de que si disponemos de ciertas características del juego que son sensibles a las trampas (*cheating*), la lógica asociada a prevenirlas deberá ser tratada de forma autoritativa (centralizada en el servidor) mientras que otras características menos importantes se pueden tratar de forma no autoritativa (en los clientes), reduciendo así el cuello de botella del servidor e incrementando el rendimiento del juego.

3.2. *Peer-to-peer*

En un sistema *peer-to-peer* (o de igual a igual), todas las instancias se encuentran en el mismo nivel jerárquico, es decir, se considera a todos los *hosts* iguales. Por lo tanto, en un juego basado en un sistema *peer-to-peer* no hay ninguna instancia que tenga más control del juego que las otras, ni existe ningún mediador que se encargue de distribuir el estado del juego o enviar mensajes entre los diferentes clientes.

Juegos *peer-to-peer*

Algunos ejemplos son *Age of Empires* o *Advance Wars*.

Algunas de las ventajas que proporciona un entorno *peer-to-peer* son las siguientes:

- No es necesario invertir en recursos como servidores específicos o técnicos de soporte, ya que cada jugador actúa como administrador de su propio equipo.

- Se implementa una sola lógica que sirve a la vez de cliente y servidor y es común para todos, lo que puede facilitar el desarrollo del juego y el diseño de su arquitectura interna.

- No se depende de un sistema central, evitando problemas de conexiones y cuellos de botella. Así, si un cliente no funciona, esto no repercute en el resto de los jugadores.

Por otro lado, los sistemas *peer-to-peer* tienen los siguientes inconvenientes:

- Son poco escalables. Se pueden utilizar sin problemas a pequeña escala, pero si intentamos utilizarlos con muchos clientes distribuidos el rendimiento generalmente se ve afectado negativamente.

- La falta de un sistema central hace que las posibilidades de desincronización entre clientes sean mucho mayores, provocando posibles incongruencias en lo que dos clientes pueden estar viendo al mismo tiempo.

- La seguridad de este tipo de sistemas es muy baja, ya que tenemos que confiar en que los otros clientes sean fiables.

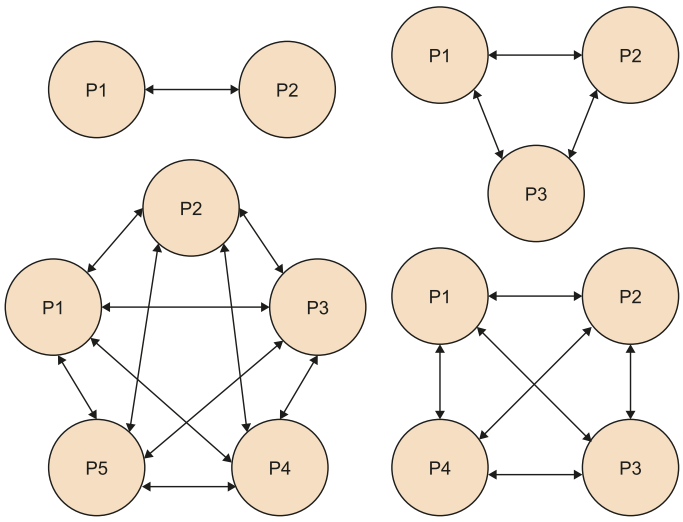
Las primeras implementaciones de sistemas *peer-to-peer* utilizaban la técnica de difusión o *broadcast* (envío simultáneo a todos los nodos) de las redes LAN. A medida que la tecnología de red fue evolucionando, estos sistemas han dejado de utilizarse, básicamente porque en Internet es inviable hacer *broadcasting* de la misma manera como se hace en las redes LAN.

Tabla 2. Correspondencias entre jugadores, tipos de red, conexiones, envíos y recepciones

Número de jugadores / Tipo de red	Broadcast	Peer-to-Peer	Cliente / Servidor
Número de conexiones	0	$\sum_{x=1}^{N-1} x$	Cliente = 1 Servidor = N
Envíos	1	N-1	Cliente = 1 Servidor = N
Recepciones	N-1	N-1	Cliente = 1 Servidor = N

Un sistema *peer-to-peer* se implementa de forma que se crean diversas conexiones equivalentes a el caso cliente/servidor entre las diferentes instancias del juego. En el caso de que el número de jugadores crezca, se tendrían que hacer demasiadas conexiones, lo que puede convertir en inviable el sistema.

Figura 4. Conexiones en un sistema *peer-to-peer*



Esto significa que en el caso de dos jugadores necesitaremos una única conexión, siendo óptimo respecto al esquema cliente-servidor. Sin entrar a comparar este escenario, si seguimos subiendo el número de jugadores, esta opción pasa a empeorar rápidamente. En el caso de tres jugadores serán dos conexiones por jugador, tres en total. Con cuatro jugadores pasamos a necesitar tres conexiones por jugador, siendo seis en total. Y ya con cinco jugadores, el número de conexiones se eleva a cuatro por jugador, pasando a ocho conexiones en global.

Muchos jugadores

En el caso de querer implementar un juego con un número de jugadores elevado, se requiere hacer uso de técnicas o estructuras que ahorren y compartan todas las conexiones posibles para minimizar este problema tanto como sea posible.

3.3. Clasificación

Recopilando los conceptos que hemos visto hasta ahora y atendiendo a la interacción entre jugadores, los videojuegos se pueden clasificar como:

- **Juegos por turnos**. Los jugadores intervienen en el juego de manera alternativa, no simultánea. En este tipo de juegos, la percepción del mundo del juego que tienen los diferentes jugadores puede ser exactamente la misma.

- **Juegos interactivos**. Los jugadores interactúan simultáneamente con el mismo mundo del juego. En este tipo de juegos, mantener la coherencia absoluta entre jugadores suele ser imposible en la práctica.

Dependiendo de la interacción de clientes con el servidor, los juegos se pueden clasificar como:

- **Juegos volátiles**. Se juegan en partidas que empiezan y acaban en un lapso determinado de tiempo. Si se cae el servidor o se corta la conexión, el jugador entiende que el progreso se pierde. No obstante, muchos juegos volátiles tienen una pequeña parte de persistencia remota para *rankings*, puntuaciones, etc.

- **Juegos en la nube**. Un servidor (o varios) actúa como repositorio de datos, gestor de mensajería y proveedor de otros servicios, dotando de persistencia a las acciones de los jugadores. El progreso del jugador debe conservarse. Normalmente se usa esta terminología para juegos por turnos, basados en datos y comandos simples.

- **Juegos masivos**. Una red de servidores que mantienen un mundo persistente y se comunican con los clientes. El progreso del jugador debe conservarse.

Para finalizar, se ejemplifican estas clasificaciones con un título muy conocido para cada una de ellas.

Tabla 3. Clasificaciones de juegos con ejemplos

	Peer-to-peer	**C/S volátiles**	**C/S en la nube**	**C/S masivos**
Por turnos	*Advance Wars*		*Clash of Clans*	
Interactivos	*Age of Empires*	*Battlefield*		*World of Warcraft*

4. Técnicas de mejora de juegos en red

Videojuegos en línea

Llamamos videojuegos en línea a aquellos videojuegos que pueden jugarse a través de Internet, independientemente de la plataforma.

En el momento en que se realiza la fase de pruebas de un videojuego en red es cuando se manifiesta una serie de factores negativos que no están presentes en un entorno minuciosamente controlado, como es una red local. Dado que el trabajo del desarrollador es anticiparse a todos estos problemas y hacer que no sean visibles para el usuario, antes de lanzar un videojuego en línea, y especialmente durante su fase *beta*, se deben solucionar los problemas conocidos y aplicar mejoras de rendimiento, así como medidas de seguridad.

4.1. Problemas en la transmisión de datos

Como ya hemos visto, los videojuegos en línea se encuentran en un ambiente típicamente hostil, ya que deben competir por el ancho de banda de cara al envío de paquetes a los servidores y los clientes se encuentran repartidos por todo el mundo. Esto produce, a diferencia de los entornos LAN, que en los juegos en línea sea frecuente enfrentarnos a la pérdida de datos y el retraso. En este apartado se exploran algunos de los problemas conocidos al desarrollar juegos en red en busca de posibles soluciones.

4.1.1. Latencia

El primero de estos factores es la latencia. La palabra *latencia* tiene diferentes significados en diferentes situaciones.

> En el contexto de los juegos de ordenador, se refiere a la cantidad de tiempo entre una causa y su efecto observable.

Dependiendo del tipo de juego, esto puede darse en situaciones muy diversas: desde el período de respuesta entre un clic del ratón y la acción de una unidad en un juego de estrategia, al período comprendido entre el momento en que un usuario realiza una acción con un dispositivo de entrada –como apretar un botón– y que el vehículo del jugador en un juego de coches realmente se mueva.

Una cierta cantidad de latencia es inevitable y diferentes géneros de juegos tienen diferentes umbrales de latencia aceptables. Por ejemplo, los juegos de realidad virtual son típicamente los más sensibles a la latencia, debido a sus estrictos requerimientos de tiempo real. En estos casos, se requiere una latencia de menos de 20 milisegundos para no producir efectos negativos en la experiencia y la salud del jugador. En juegos de lucha, acción en primera persona u otros juegos de acción rápida también hay una alta sensibilidad a la latencia,

pudiendo variar de 16 a 150 milisegundos antes de que el usuario empiece a notar, independientemente de la velocidad de fotogramas, que el juego es lento o no responde correctamente.

Por lo tanto, la disminución de la latencia es una buena forma de mejorar la experiencia de juego y, para ello, se requiere entender los factores que la afectan.

Entre otras causas, el retraso experimentado por un paquete a medida que viaja desde un *host* de origen a su destino suele la principal fuente de latencia en juegos multijugador. Podemos separar en cuatro clases principales el retraso que experimenta un paquete durante su vida útil:

NAT (*Network Address Translation*)

Es un mecanismo utilizado por enrutadores IP para intercambiar paquetes, basado en convertir en tiempo real las direcciones utilizadas en los paquetes transportados.

- **Retraso de procesado**. Un enrutador se encarga de leer los paquetes de la interfaz de red, examinar la dirección IP de destino, resolver la máquina destino que debe recibir el paquete y finalmente transmitirlos por la interfaz adecuada. Este procesamiento implica un retraso que puede ser incrementado al incluir cualquier funcionalidad adicional que el enrutador proporcione, como NAT (*Network Address Translation*) o cifrado.

- **Retraso de transmisión**. Para que un enrutador transmita un paquete, debe tener una interfaz de capa de enlace que permita que este se transmita a través de algún medio físico. El protocolo de capa de enlace controla la velocidad media a la que los bits se pueden escribir en el medio, implicando un tiempo necesario para realizar dicha operación.

- **Retraso de espera**. Un enrutador solo puede procesar un número limitado de paquetes a la vez. Si los paquetes llegan a un ritmo más rápido que el de procesado, entran en una cola de recepción, esperando para ser procesados. Del mismo modo, una interfaz de red solo tiene salida para un único paquete a la vez. Así pues, después de que un paquete sea procesado, si la interfaz de red correspondiente está ocupada, se entra en una cola de transmisión.

- **Retraso de propagación**. En su mayor parte, independientemente del medio físico, la información no puede viajar más rápido que la velocidad de la luz. Con fibra óptica se consigue más de la mitad de la velocidad de la luz, pero no la velocidad completa. Conexiones a través del «aire», como las de las redes móviles o las de satélite, tienen una latencia muy elevada.

Algunos de estos retrasos pueden ser paliados mediante ciertas mejoras. Además, el retraso de procesado es típicamente un factor menor. En la actualidad, la mayoría de los procesadores de enrutadores son muy veloces.

Ping

Es una utilidad diagnóstica en redes de computadoras que comprueba el estado de la comunicación entre dos puntos remotos y se utiliza para medir la latencia entre ellos.

Estas mejoras pueden ser medidas de bajo nivel, como la reducción de cabeceras en el tamaño de los paquetes para reducir el retraso de transmisión, o reducir al mínimo el procesado para ayudar a minimizar las colas de retraso, o medidas de alto nivel, más relacionadas con las decisiones que podamos tomar como desarrolladores de videojuegos.

El retraso de propagación es a menudo un buen objetivo para la mejora de la latencia, debido a que se basa en la distancia entre *hosts*. La mejor manera de optimizar sería situar más cerca a los anfitriones (*locality*) dependiendo de la topología.

Matchmaking

Llamamos *matchmaking* al proceso de agrupar y enfrentar jugadores en sesiones de juego en línea.

- En los juegos *peer-to-peer*, esto significa dar prioridad a la localidad geográfica en el momento del *matchmaking*.

- En los juegos cliente-servidor, esto significa desplegar servidores de juego seguro que estén disponibles cerca de los distintos jugadores.

A veces, la localización física no es suficiente para garantizar el retraso de baja propagación: puede que no existan conexiones directas entre localizaciones, lo que requiere enrutadores para encaminar el tráfico por rutas tortuosas en lugar de a través de una línea directa. Es importante tener en cuenta las rutas existentes y futuras en la planificación de las ubicaciones de los servidores de juego.

Sin embargo, en algunos casos no es factible dispersar los servidores de juego a lo largo de un área geográfica, ya que se desea que todos los jugadores de todo un continente puedan jugar entre ellos. Casos como *League of Legends* se encontraron en esta situación: la dispersión de servidores de juego por todo el país no era una opción y decidieron ejecutar la estrategia inversa. Para ello, construyeron su propia infraestructura de red, analizando los proveedores de servicios de Internet (ISP) a lo largo de los Estados Unidos, para asegurar que podían controlar las rutas de circulación y reducir la latencia de la red tanto como fuera posible.

Debido a que la *latencia* es un término muy utilizado, los desarrolladores de juegos suelen discutir con mayor frecuencia el concepto específico del tiempo de ida y vuelta (*Round Trip Time*, RTT) para condensar en una única métrica los cuatro tipos de retraso indicados.

RTT (*Round Trip Time*)

Se refiere al tiempo que tarda un paquete en viajar de un *host* a otro, y luego el paquete de respuesta en recorrer todo el camino de vuelta.

Esto termina reflejando no solo las demoras de procesado, gestión de colas, transmisión y propagación de dos rutas, sino también en los FPS (*frames per second*) del receptor, es decir, que reduce la velocidad de fotogramas de la máquina remota, ya que también afecta al proceso de envío de paquetes de respuesta. Además, el tráfico no tiene necesariamente la misma velocidad de viaje en cada dirección. El RTT rara vez se duplica a partir del tiempo que tarda un

paquete a pasar de un huésped a otro, ya que el protocolo TCP/IP no garantiza que el camino sea el mismo para ambos trayectos. A pesar de ello, es práctica común usar como aproximación el cálculo de la mitad del RTT como tiempo de viaje de una sola ruta.

4.1.2. *Jitter*

Con una buena estimación del RTT ya se pueden tomar medidas para mitigar estos retrasos y dar a los jugadores la mejor experiencia posible. Sin embargo, cuando se escribe código de red, se debe tener en cuenta que el RTT no es ni mucho menos un valor constante. Para cualquiera de los dos *hosts*, el RTT entre ellos no se cierne típicamente alrededor de un valor determinado sobre la base de los retrasos medios involucrados.

Se conoce como *jitter* la variabilidad del tiempo de llegada de dos paquetes consecutivos. Este efecto es especialmente molesto en los juegos multijugador en línea, ya que provoca que algunos paquetes lleguen demasiado pronto o tarde, impidiendo entregarlos a tiempo.

Cualquiera de los retrasos involucrados puede cambiar con el tiempo, dando lugar a una desviación en RTT del valor esperado y contribuir a la fluctuación, aunque algunos son más propensos a producir variaciones que otros:

- **Retraso de procesamiento**. Al ser el componente menos significativo de la latencia de red también es el menos significativo, contribuyendo a su desfase. Puede variar en cómo los enrutadores ajustan las rutas dinámicamente al viajar los paquetes, pero es una preocupación menor.

- **Retrasos de transmisión y propagación**. Están muy relacionados, dado que los protocolos de capa de enlace determinan el retraso de la transmisión y la longitud de ruta determina el retraso de propagación. Así pues, estos cambian cuando los enrutadores equilibran de forma dinámica la carga de tráfico y alteran las rutas para evitar las zonas congestionadas. Esto puede fluctuar rápidamente durante los momentos de tráfico pesados, con lo que los cambios de ruta pueden alterar significativamente los tiempos de ida y vuelta.

- **Retraso de espera**. Está relacionado con el número de paquetes que un enrutador puede procesar. A medida que el número de paquetes que llega a un enrutador varía, el retraso de cola también varía. Las ráfagas de tráfico pesado pueden causar demoras de espera significativas y variar los RTT.

Además, el *jitter* puede afectar negativamente a los algoritmos de reducción de RTT, y lo que es peor, puede causar que los paquetes lleguen completamente desordenados.

4.1.3. Pérdida de paquetes

Más importante que la latencia, el RTT y el *jitter* es la pérdida de paquetes, el mayor problema con el que se enfrentan los desarrolladores de juegos de red. Claramente ya no es un problema de que un paquete tarde mucho tiempo en llegar a su destino, sino que el paquete no llegue nunca.

Los paquetes pueden perderse por diversas razones.

- **Debido a un medio físico poco fiable.** En su origen, la transferencia de datos es la transferencia de energía electromagnética. Cualquier interferencia electromagnética externa puede provocar la corrupción de estos datos. En el caso de los datos corruptos, la capa de enlace detecta la corrupción al validar las sumas de comprobación (por ejemplo, CRC) y descarta las tramas que contienen errores.

CRC (*Cyclic Redundancy Check*)

Es un código de detección de error cuyo cálculo es una división en el que el resto debe coincidir.

- Las capas de enlace tienen reglas sobre cuándo pueden y no pueden enviar datos. A veces, un canal de capa de enlace está completamente lleno, y una trama saliente debe descartarse. Debido a que la capa de enlace no proporciona ninguna garantía de fiabilidad, esta es una respuesta perfectamente aceptable.

- **Una capa de red no fiable**. Recordemos que cuando los paquetes llegan a un enrutador, se colocan en una cola de recepción. Esta cola tiene un número máximo de paquetes que puede contener. Cuando la cola está llena, el enrutador comienza a descartar tanto paquetes ya en cola como paquetes entrantes. Por lo tanto, la pérdida de paquetes perdidos es un hecho que no se puede evitar, y se debe diseñar su arquitectura de red teniendo en cuenta este problema.

Independientemente de la mitigación de la pérdida de paquetes, la experiencia de juego será mejor con un menor número de paquetes perdidos. Una vez más, la solución de utilizar un centro de datos con servidores tan cerca de sus jugadores como sea posible es la solución más directa y fiable. Esto es debido a que implica un menor número de enrutadores y cables, y por lo tanto, una menor probabilidad de que uno de ellos descarte nuestros datos.

Otra aproximación consiste en enviar el menor número de paquetes que podamos. Muchos enrutadores calculan su capacidad de la cola en función del recuento de paquetes, no en la cantidad de datos totales. En estos casos, un juego tiene una mayor probabilidad de inundar enrutadores y desbordar las colas si envía muchos paquetes pequeños que si lo hace con un menor número de paquetes grandes. El envío de diez paquetes de 100 bytes a través de

un enrutador obstruido requiere diez ranuras libres en la cola para evitar la pérdida de paquetes. Sin embargo, el envío de los mismos 1.000 bytes en un solo paquete solo requiere una ranura de cola libre.

Sin embargo, no todos los enrutadores basan la adjudicación de sus ranuras en el número de paquetes; algunos de ellos se basan en el ancho de banda entrante y entonces resulta más beneficioso enviar paquetes pequeños.

Cuando sus colas están llenas, un enrutador no necesariamente deja caer cada paquete entrante, sino que en vez de eso puede descartar un paquete que aguarda en cola. Esto sucede cuando el enrutador determina que el paquete entrante tiene mayor prioridad o es más importante que el de la cola. Los enrutadores toman decisiones de prioridad en base a datos de calidad de servicio (*Quality of Service*, QoS) que se encuentran en la cabecera de la capa de red, o también a partir de información obtenida mediante la inspección de la carga útil del paquete.

Algunos enrutadores están configurados incluso para tomar decisiones rápidas y destinadas a reducir el tráfico en general. Por ejemplo, a veces se dejan caer los paquetes UDP antes que los paquetes TCP porque estos acaban siendo enviados de forma automática.

La comprensión de las configuraciones del enrutador en torno a los centros de datos, y de los ISP en todo el mercado de destino, puede ayudar a ajustar los tipos de paquetes y los patrones de tráfico para reducir la pérdida de paquetes. En última instancia, la forma más sencilla de reducir la pérdida de paquetes es asegurarse de que los servidores usan conexiones a Internet rápidas, estables y que están tan cerca de sus clientes como sea posible.

4.2. Mejora del rendimiento

Una vez estudiados los problemas, vamos a proceder a describir las soluciones para mejorar el rendimiento en los videojuegos en línea.

4.2.1. A nivel de infraestructura

La primera opción que tenemos para mejorar nuestro sistema es a nivel de hardware, sistema operativo y conexión a Internet. Es muy importante que nuestro sistema esté preparado para poder enviar y recibir todos los datos que necesitamos. Por ello, es necesario utilizar una conexión que nos permita mejorar algunos de estos aspectos:

- Reducir al mínimo el número de paquetes perdidos.

- Reducir al mínimo el tiempo de latencia, es decir, el tiempo que tarda un paquete en viajar entre el origen y el destino.

- Reducir al máximo el *jitter*, es decir, la diferencia entre el tiempo de llegada de dos paquetes consecutivos. Nos interesa mantener un ritmo constante para poder integrar el flujo de entrada/salida coherentemente en el juego.

Estas mejoras se realizan normalmente a base de invertir más dinero en infraestructura, ya que no se pueden realizar mediante programación.

4.2.2. A nivel de protocolo

Otro elemento clave para mejorar la fluidez de un juego es la optimización del protocolo de comunicaciones utilizado. Vamos a dar algunos consejos sobre cómo puede mejorarse el tamaño de los paquetes y la cantidad de paquetes que se envían:

- Enviar únicamente los cambios de estado en lugar de enviar continuamente el estado actual. Por ejemplo, en lugar de enviar nuestra posición continuamente, solo enviamos la orden de avanzar, retroceder, etc. Este proceso requiere que cada cierto tiempo se envíe algún paquete de sincronización para garantizar que ambos sistemas se encuentran en el mismo estado.

- Elegir un área alrededor del personaje y enviar únicamente aquellos eventos que ocurran dentro de esta zona. También podemos establecer diferentes secciones más grandes en las que podemos enviar menos información (por ejemplo, solo posiciones de personajes para dibujar el mapa).

- Intentar condensar el máximo de información en un solo paquete de datos. Más paquetes implican un mayor *overhead* (ya que enviamos más cabeceras), y por tanto generamos más tráfico de red del que es necesario.

- Mensajería jerárquica. Se trata de adaptar la cantidad de información enviada según las diferentes velocidades de los clientes (podemos tener desde líneas de fibra o ADSL de todo tipo de velocidades y simetrías). La técnica consiste en asignar una cierta prioridad a los mensajes que vamos a enviar y decidir si lo transmitimos o no dependiendo de la importancia que tenga para el correcto funcionamiento del juego. Por ejemplo, los cambios de posición tienen una prioridad muy alta, pero un cambio en la animación del usuario es de prioridad muy baja.

4.2.3. A nivel predictivo

Otro de los métodos más usados para que el jugador no note los problemas que pueda haber en la red es el uso de la predicción del comportamiento de los elementos del juego.

De esta forma, el cliente intenta predecir cuál es el próximo movimiento que van a realizar los objetos que se encuentran en su zona. Cada vez que recibe del servidor una posición y una orientación, simplemente hemos de corregir parcialmente la posición de los objetos que habíamos intuido para que se ajusten a la realidad. Es recomendable utilizar algoritmos que hagan esta transición de forma simple, para que no veamos cambios bruscos en las posiciones de los elementos.

Figura 5. Algoritmo de predicción en *Counter-Strike*

Un algoritmo de predicción necesita dos componentes principales:

- los datos históricos de qué han hecho todos los elementos en los últimos informes del servidor, y

- un buen sistema de predicción que permita anticipar el movimiento real de los objetos.

Aun así, no hay ningún sistema que sea capaz de predecir siempre las acciones de los otros usuarios y los elementos del sistema, por lo que debemos tener claro que el resultado de nuestra predicción no es nunca fiable al cien por cien.

En algunos juegos en línea, la predicción puede dar lugar a situaciones bastante extrañas o cómicas. Aunque la sensación de que estamos jugando es bastante continua, si ha habido un corte en la red puede ser que aparezcamos de repente en un punto donde nos encontrábamos hace un buen rato. Esto

es debido a que, cuando se recupera la conexión, vemos que existe una gran diferencia entre el estado del cliente y el del servidor, con lo que el servidor nos vuelve a situar en la ultima posición válida que conocía.

En algunos juegos este problema puede ser más preocupante, ya que si la conexión se corta en un momento clave (en un combate, por ejemplo), cuando el servidor nos actualiza el estado podemos descubrir que nuestro personaje ha muerto. En estos casos de poco nos sirve la predicción de las acciones.

4.2.4. En el envío de acciones

En algunos tipos de juegos, sobre todo los que requieren mucho movimiento de elementos (como los RTS), no es viable enviar en cada momento las posiciones y el estado de todos los elementos, ya que esto conllevaría un gasto de ancho de banda considerable. En su lugar, se envían las órdenes que dan los jugadores y a qué elementos del juego se las dan.

Para que este sistema funcione, todos los clientes tienen que ejecutar la misma versión del juego, a la misma velocidad de ejecución (se acostumbra a trabajar a 10fps) y esta debe ser totalmente determinista. Es decir, si un jugador da una orden a un comando de mover 100 unidades de A a B, todos los clientes deben calcular la misma ruta para las 100 y empezar a moverlos al mismo momento.

Garantizar el determinismo

Uno de los problemas más importantes que hay que solucionar para garantizar el determinismo es la resolución de los números de coma flotante.

El elemento clave en el sistema es la sincronización inherente. Aunque no se envían mensajes, todos los sistemas deben estar sincronizados en todos los ciclos de ejecución. Cada cierto tiempo se debe llevar un testeo de sincronización entre todos los elementos del juego (por ejemplo, comprobar que todas las unidades están en la misma posición y/o en el mismo estado). Un solo fallo de sincronización no tiene solución y conllevará la finalización de la partida.

4.3. Seguridad

La seguridad en la comunicación es necesaria para garantizar el correcto desarrollo de un juego. La seguridad es importante por tres razones principales:

- Para proteger la información sensible de los usuarios (contraseñas, etc.).
- Para evitar suplantaciones de personalidad (un usuario se hace pasar por otro).
- Para proporcionar un entorno de juego justo, en el que ningún usuario pueda aventajar a los otros utilizando trucos o trampas.

La protección de la conexión se puede realizar mediante diversas técnicas que actúan en diferentes niveles de la arquitectura TCP/IP:

- En cuanto a la red o al transporte, podemos filtrar paquetes sospechosos o maliciosos. En este ámbito podemos detectar y prevenir algunos de los ataques de *hackers* que intentan modificar direcciones y puertos para engañar a los servidores.

- En cuanto a la aplicación, podemos proteger los datos cifrando la información contenida en los paquetes. Solo es recomendable aplicar esta técnica en momentos puntuales y cuando se trate de información sensible, ya que el proceso de cifrado y descifrado requiere cierta computación que afectaría al desarrollo de una partida.

Existen servicios *online* que proporcionan mecanismos para mejorar la seguridad de los juegos y evitar que los usuarios utilicen trampas para sacar ventaja respecto a otros usuarios.

En este sentido, si optamos por usar una plataforma de servicios, algunos de los que se ofrecen son proteger los videojuegos de algunas de las trampas más conocidas, como por ejemplo, el falseo de estadísticas o marcadores. Estas plataformas ofrecen un entorno cerrado en el que existe un control más exhaustivo de los juegos que están soportados.

Adicionalmente, este tipo de servicios mantienen una base de datos sobre usuarios «baneados» (*banned*), correspondiente a los jugadores a los que se ha descubierto intentando realizar alguna acción ilegal y, en consecuencia, ya no se les permite acceder al juego, como castigo.

5. Desarrollo en Unity

El motor de juego Unity fue lanzado por primera vez en 2005. En los últimos años se ha convertido en un motor de juego muy popular utilizado por muchos desarrolladores. Este motor ofrece algunas funciones de sincronización y RPC muy útiles. Concretamente, a partir de Unity 5.1 se introdujo una nueva biblioteca de red llamada UNet.

5.1. UNet

La biblioteca UNet proporciona dos API diferentes:

- HLAPI (*high-level API*), de nivel superior, capaz de manejar la mayoría de los casos de uso de un videojuego en red a nivel de objetos de juego y sus interacciones.

- LLAPI (*low-level API*), de transporte de bajo nivel, por debajo de la anterior, está pensada para ser utilizada como capa de comunicación de forma muy similar al mecanismo de *sockets*.

Vamos a describir con mayor detalle la API de nivel superior, siendo LLAPI relegada a una forma avanzada de afinar los mecanismos de red en caso de que esto fuese necesario.

5.1.1. UNet HLAPI

En Unity, cada clase *GameObject* se usa como recipiente para diversos componentes, con todos los comportamientos delegados. Esto permite una mejor delimitación entre diferentes aspectos del comportamiento de un objeto de juego, aunque si hay dependencias entre múltiples componentes hace que el desarrollo en red de estos sistemas sea aún más complejo.

En esta API de alto nivel, Unity proporciona un administrador de red (la clase de ***NetworkManager***) para encapsular el estado de un juego en red. Este administrador de red puede funcionar en tres modos diferentes: como cliente independiente, como servidor independiente (dedicado), o como ***host***, combinando la idea de cliente y servidor al mismo tiempo.

Debido a que Unity utiliza la topología cliente-servidor, el mecanismo de *spawn* es muy diferente al mecanismo análogo en un videojuego de un solo jugador en local. En concreto, cuando un objeto es generado en el servidor a través de la función ***NetworkServer.Spawn***, significa que este objeto de juego va a ser rastreado por el servidor mediante un identificador de red.

Spawn

Es un mecanismo de instanciación dinámica que permite crear *GameObjects* en tiempo de ejecución a partir de un *prefab*.

Además, este objeto debe ser replicado y generado en todos clientes al mismo tiempo. Para poder llevar a cabo esta operación, es requerimiento indispensable registrar el *prefab* (configuración preestablecida) del *GameObject* implicado. Una vez el objeto es generado en el servidor, las propiedades pueden ser replicadas en los clientes de diferentes formas. Otro requerimiento en este sentido es que el *GameObject* debe heredar de ***NetworkBehaviour***, en vez del habitual *MonoBehaviour*.

Una vez llegamos a este punto, la forma más simple de replicar es anotar cada variable del *GameObject* a sincronizar mediante el atributo **[*SyncVar*]**. Este mecanismo funciona con todos los tipos proporcionados por Unity. Por lo tanto, cualquier variable de este grupo de *SyncVars*, al recibir un nuevo valor, lo replica automáticamente en todos los clientes.

Es por ello por lo que, cuando se usa una estructura propia como variable, hay que tener en cuenta que todo el contenido de esta se envía de golpe. Por lo tanto, si esta es muy grande y solo cambia una pequeña parte de ella, usaremos un ancho de banda excesivo.

Si necesitamos hacer uso de un sistema de control más refinado a la hora de replicar variables, podemos sobrescribir las funciones *OnSerialize()* y *OnDeserialize()* para leer y escribir manualmente estas variables. Esto significa añadir una funcionalidad personalizada, pero no se permite su uso combinado con *SyncVars*, teniendo que elegir uno de los dos métodos.

Unity también implementa el mecanismo de llamadas remotas. En concreto, Unity proporciona un mecanismo llamado ***Command***, representando una acción enviada de un cliente al servidor, restringido a los objetos controlados por el jugador. Por el contrario, una función de tipo **Client RPC** es una acción enviada desde el servidor al cliente.

Nota

Al igual que con *SyncVar*, estos tipos de funciones RPC solo son soportados por subclases de *NetworkBehaviour*.

El sistema para activar los mecanismos con estas funciones especiales es muy similar al mecanismo de sincronización de variables, requiriendo el atributo **[*Command*]** y el prefijo *Cmd* en cada función, como por ejemplo *CmdFire*(..). Y de forma análoga, anotando con el atributo **[*ClientRpc*]** y nombrando con el prefijo *Rpc* activamos el otro tipo de llamada remota. De este modo, cualquiera de estas funciones renombradas y anotadas puede ser llamada por medio de una función local y esta ser automáticamente ejecutada de forma remota.

5.1.2. UNet LLAPI

La API de la capa de transporte proporcionada por UNet es una fachada de acceso a los *sockets*. Lógicamente, esta capa de bajo nivel puede ser vista como una pila de protocolos de red integrada en la parte superior de la capa de transporte, que contiene una capa de red y una capa de transporte.

LLAPI

El diseño de la LLAPI se asemeja mucho a la API de *sockets* BSD.

Para la capa de transporte, en lugar de solicitar específicamente una conexión UDP o TCP, se debe especificar cómo desea utilizar la conexión. Es decir, se puede crear un canal de comunicación y solicitar diferentes valores relacionados con la calidad del servicio (QoS):

- ***Unreliable***. Envía mensajes sin ninguna garantía de recepción.

- ***UnreliableSequenced***. Envía mensajes sin garantizar la llegada, pero los mensajes fuera del orden son descartados. Especialmente útil en comunicación por voz.

- ***Reliable***. Se garantiza la recepción de los mensajes, siempre y cuando se mantenga la conexión.

- ***ReliableFragmented***. Un mensaje fiable que puede ser fragmentado en diversos paquetes. Esto es útil en el envío de ficheros de gran tamaño a través de la red, que deben ser reensamblados en destino.

La capa de red se utiliza para crear conexiones entre máquinas, entrega de paquetes y el control de flujo y la congestión.

Las conexiones se pueden establecer a través de la llamada de función ***NetworkTransport.Connect***. Esta llamada retorna un identificador de conexión, que puede ser utilizado como parámetro en otras funciones de la clase ***NetworkTransport*** para enviar y recibir paquetes, así como desconectarse.

Al igual que los *sockets*, la LLAPI soporta solo una abstracción para el intercambio de mensajes binarios sin formato. No proporciona mecanismos de alto nivel como serialización, codificación, llamadas RPC, etc.

5.2. Alternativas a UNet

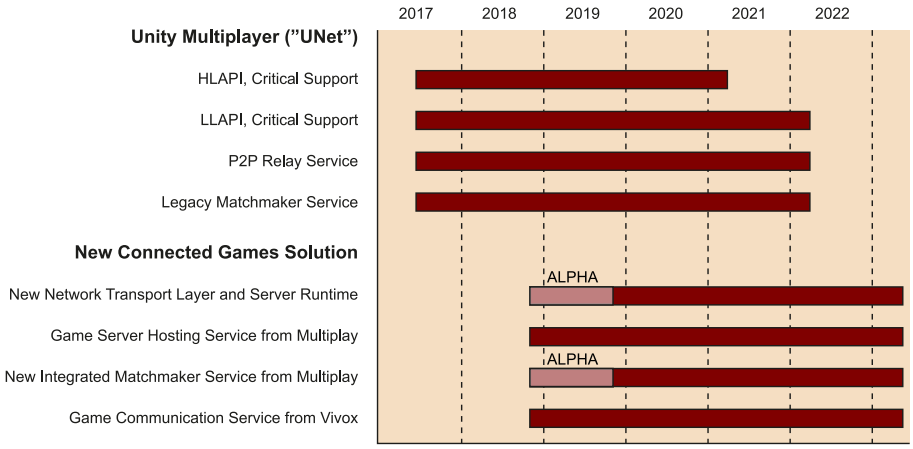
El 2018 Unity decidió que dejaba como obsoleta (*deprecated*) la biblioteca de UNet porque no cumplía las necesidades mínimas de muchos de los creadores de videojuegos multijugador actuales. Estas necesidades se traducen básicamente en fiabilidad, rendimiento, escalabilidad y seguridad; características que UNet dejó de cumplir con el paso de los años.

Deprecation

El hecho de desaconsejar el uso de algunas características típicamente porque han sido reemplazadas o ya no se consideran eficientes o seguras, sin eliminarlas completamente para ser discontinuadas en el futuro.

Sin embargo, la alternativa a las API de UNet se ha demorado bastante y, aunque las antiguas API están soportadas hasta las versiones de Unity 2018.4 (LTS) en el caso de HLAPI y 2019.4 (LTS) en el caso de LLAPI, las nuevas versiones de la nueva API de red de Unity aún no se han liberado.

Figura 6. *Deprecation plan* para UNet



Mientras tanto, muchos desarrolladores de juegos multijugador en Unity han optado por utilizar otras soluciones, como por ejemplo ***Mirror***, que proporciona una API muy similar a la HLAPI de UNet, resuelve los errores que tenía y además está preparada para juegos multijugador masivos en línea (MMO). Esta librería está muy extendida dentro de la comunidad y, al tratarse de una API de alto nivel al igual que HLAPI, está focalizada en facilitar el desarrollo de videojuegos multijugador y no se centra en la gestión del transporte, propiamente dicho. Asimismo, cuenta con una comunidad muy activa y es un proyecto de código abierto, que asegura que el soporte esté garantizado.

6. Proyecto: *Tanks!* LAN

Como conclusión principal a todo lo que hemos visto hasta ahora, podemos decir que el multijugador en red es inherentemente complejo y requiere tener en cuenta muchos detalles que afectan directa e indirectamente a los videojuegos, así como problemas y dificultades particulares asociados con la sincronización y la comunicación entre varias instancias de un videojuego que a menudo se ejecutan en máquinas diferentes que podrían estar en partes del mundo distantes entre sí.

En este proyecto continuaremos con el ejemplo del juego de tanques, donde cada jugador controla a un tanque y dispara proyectiles para acabar con sus rivales, que ya hemos trabajado como multijugador local. Sin embargo, siguiendo la norma general, para desarrollar un multijugador en red, debemos empezar de nuevo, obviamente partiendo de los recursos (*assets*) básicos ya disponibles (*scripts* de control, modelos, *shaders*, etc.) pero reimplementando bastante lógica del juego con los mecanismos de red necesarios.

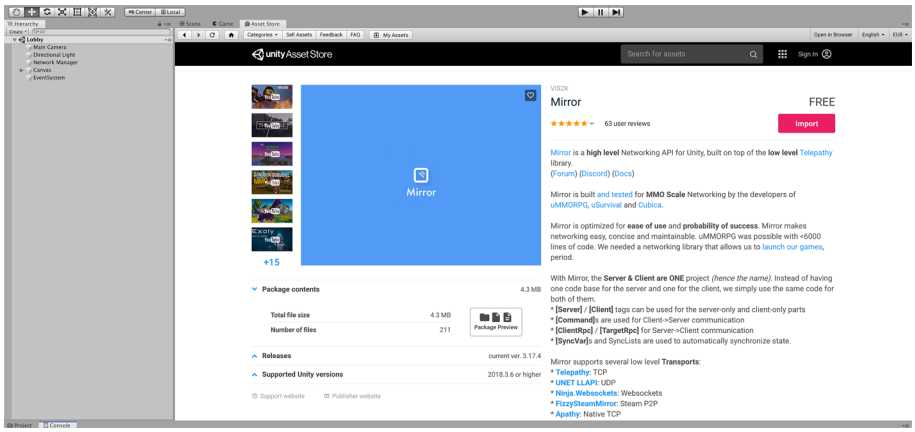
Con este proyecto intentaremos trabajar de la forma más simple posible todos los mecanismos y aspectos que permite la API ***Mirror***. A través de esta, se detallará paso a paso su funcionamiento, planteando una serie de retos para mejorar el juego inicial. Aunque se trate de un *shooter* multijugador (uno de los géneros más populares), el uso de cada mecanismo se puede extrapolar a cualquier género y a muchas situaciones diversas pero comunes del resto de juegos multijugador.

Así pues, se tratará de un proyecto cliente/servidor al que podrán unirse varios jugadores (tanques) en red y dispararse proyectiles que afectarán a su salud según los impactos recibidos. Veremos cómo implementar el disparo de proyectiles con *spawnable prefabs*, la utilización de un *authoritative server* para controlar la salud de los distintos jugadores y cómo implementar las rutinas de muerte y reaparición mediante llamadas RPC.

6.1. Instalación de *Mirror*

El primer paso será obtener el *asset* de ***Mirror*** desde el *asset store* de Unity e importarlo a nuestro proyecto. Una vez hecho esto es imprescindible que reiniciéis Unity para que los componentes del menú se actualicen correctamente.

Figura 7. Importación del *asset* ***Mirror*** en el proyecto

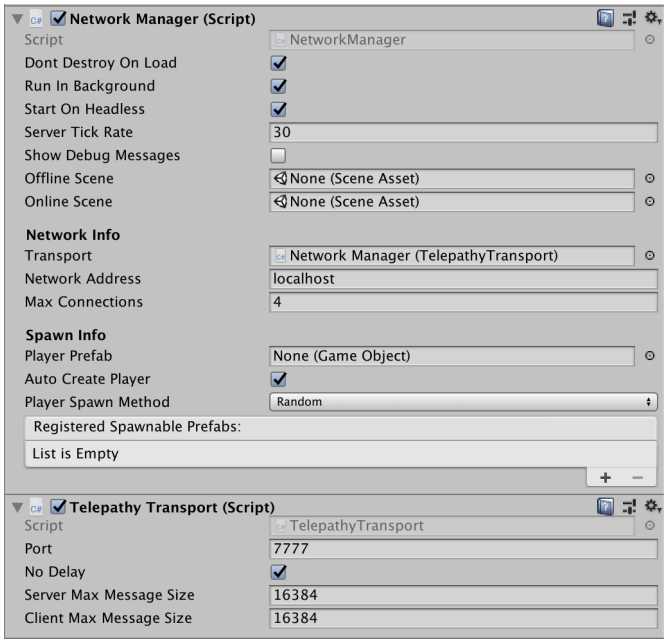


6.2. Creación del administrador de red

Debemos crear un nuevo administrador de red (clase *NetworkManager*). Este administrador de red controlará el estado del juego multijugador, incluyendo la gestión del estado del juego, del *spawn*, gestión de escenas y permitiendo el acceso a la información de *debug*.

Para crear un nuevo administrador de la red, basta añadir un componente de tipo *NetworkManager* a un *GameObject* llamado, por ejemplo, *Network Manager*. A continuación, podemos ver una captura del componente *NetworkManager* y hacernos una idea rápida de cómo gestiona el estado de la red del juego.

Figura 8. *Script Network Manager*



Como podemos observar, este componente permite configurarse de la siguiente forma:

- Evitar su destrucción al cargar una escena.
- Ejecutarse en segundo plano.

- Ejecutarse en modo *headless* si el dispositivo no posee de capacidades gráficas.
- Frecuencia (*tick rate*) a la que un servidor actualiza el estado del juego. Solo se usa en modo *headless*.
- Mostrar o no mensajes de depuración.
- Referenciar la escena de *lobby* asociada al juego.
- Referenciar la escena de juego asociada al juego.
- Protocolo de transporte a utilizar (por defecto, *Mirror* utiliza las librerías *Telepathy*), dirección de red y número máximo de clientes por servidor.
- Información sobre la generación de objetos de *spawn*.
- Configuración avanzada del protocolo de transporte *Telepathy*.

6.3. Creación del jugador

El activo prefabricado (*prefab*) Player representará al tanque del jugador, o jugadores. De forma predeterminada, el componente *NetworkManager* instancia un *GameObject* para cada jugador que se conecta a la red, mediante la clonación y *spawn* del *prefab* Player en el juego. El *GameObject* del jugador en la escena debe tener el siguiente aspecto.

Figura 9. Aspecto del *GameObject* del jugador

Para identificar al jugador como un *GameObject* único en el juego en red, debemos añadir un componente ***NetworkIdentity*** en el *prefab* del jugador (*Player*).

Figura 10. *Network Identity*

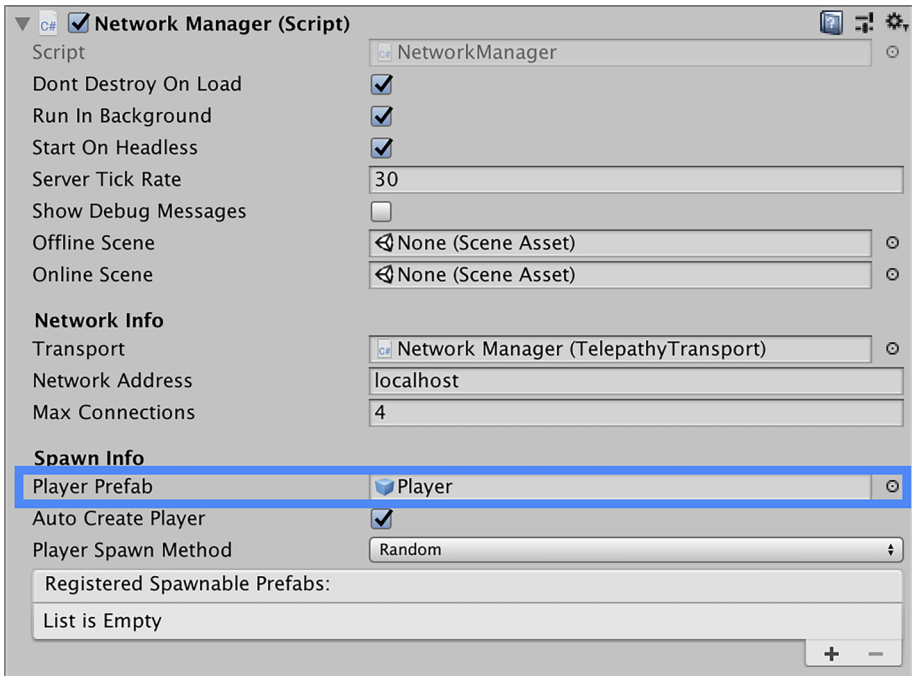
El componente *NetworkIdentity* se utiliza para identificar el objeto en la red y hacer que el sistema de red sea consciente de ello. En este componente, debemos tener la propiedad Local Player Authority activada para habilitar el control del movimiento del *GameObject* del tanque al jugador cliente.

Ved también

Para obtener más información sobre el componente *NetworkIdentity*, se puede consultar la API de *Mirror*.

Una vez que tenemos listo el *prefab* Player, este debe ser registrado en el sistema de Networking. Para ello, y como la mayoría de los juegos tienen un solo e idéntico *prefab* que representa a cualquiera de los jugadores en el juego, en nuestro caso debemos indicar el *prefab* Player en la propiedad del *NetworkManager* específica para ello (*Player Prefab*).

Figura 11. *Network Manager*



Player

Network Manager utilizará este *prefab* para hacer *spawn* de nuevos *GameObjects* del jugador en la escena.

El *GameObject* que representa el jugador se crea siempre que un nuevo jugador cliente se una a la partida del anfitrión. En este punto surgen otros temas muy relacionados con el *spawning* por red y otros detalles de la sincronización de *GameObjects* de jugador entre clientes y servidor. Pero no los trataremos hasta más adelante, una vez hayamos realizado el resto de los pasos básicos.

6.4. Creación del controlador del jugador

Control de la entrada

Por defecto, *Input.GetAxis("Horizontal")* y *Input.GetAxis("Vertical")* permiten al jugador moverse con las teclas WASD o las teclas de dirección, un *gamepad* u otro dispositivo de entrada.

Una vez tenemos las bases para la gestión de los objetos en el juego en red, pasamos a centrarnos en los aspectos vinculados a la mecánica del juego en sí. Un primer paso es poder mover el *GameObject* del tanque jugador en la escena. Para ello contamos con el *script TankController*, que sería el siguiente sin ningún código de red.

```
using UnityEngine;


public class TankController : MonoBehaviour
        void Update() {
        var x = Input.GetAxis("Horizontal") * Time.deltaTime * 150.0f;
        var z = Input.GetAxis("Vertical") * Time.deltaTime * 3.0f;


        transform.Rotate(0, x, 0);
        transform.Translate(0, 0, z);
    }
}
```

Como vemos, se trata del código típico del método *Update()*, donde se aplican las rotaciones y traslaciones del jugador local que controla el tanque. Para añadir la funcionalidad de red al movimiento del jugador y asegurarse de que solo el jugador local puede controlar el *GameObject* local, tenemos que tener en cuenta las siguientes premisas:

- Utilizar el *namespace Mirror*.

```
using Mirror;
```

Namespace Mirror

El *namespace Mirror* proporciona todo el código de red que necesitamos para escribir nuestros *scripts*.

- *Extender* de *NetworkBehaviour*, en vez de *MonoBehaviour*.

```
public class TankController : NetworkBehaviour
```

NetworkBehaviour

Los *scripts* añadidos en un *GameObject* que necesitan funciones de red deben heredar de *NetworkBehaviour*, que a su vez extiende *MonoBehaviour*.

- Se debe añadir una comprobación de la propiedad *isLocalPlayer* en la función *Update*, para que solo el jugador local la realice

```
if (!isLocalPlayer) {
     return;
}
```

Este punto se debe a que, en un proyecto en red, el servidor y todos los clientes están ejecutando el mismo código con las mismas secuencias de comandos en los mismos *GameObjects* a la vez. Además, todos estos *GameObjects* jugador fueron generados a partir de un mismo *prefab* Player y todos ellos ejecutan el mismo *script TankController*.

Ved también

La clase base *NetworkBehaviour* proporciona otras funcionalidades muy útiles. Para más información, consultad la API oficial de *Mirror*.

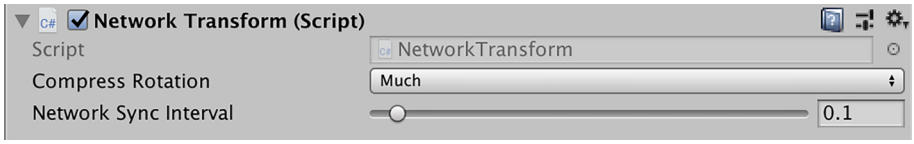
Precisamente para evitar que la secuencia de comandos sea exactamente igual, hacemos uso del concepto *LocalPlayer*, que se refiere al *GameObject* del jugador en propiedad del cliente local.

La propiedad *isLocalPlayer* se establece automáticamente a través del *NetworkManager* siempre que un cliente se conecta al servidor y nuevos *GameObjects* jugador se crean a partir del *prefab* Player. Cuando un cliente se conecta al servidor, la instancia creada en el cliente local se marca como *LocalPlayer*. Todos los demás *GameObjects* jugador que representan a este jugador (en el servidor o en otro cliente) no están marcados con esta propiedad.

Si ejecutamos el proyecto en este punto, los *GameObjects* jugador no permanecen sincronizados a través de todas las instancias del proyecto en curso. Los controles de entrada para el movimiento solo se procesan en el jugador *GameObject* local, pero la posición y la rotación de ese jugador no se actualizará en el resto de las instancias del juego.

Para mantener los *GameObjects* de los jugadores en sincronía, hemos de añadir un componente llamado ***NetworkTransform*** en el *prefab* Player. Este componente precisamente se encargará de sincronizar el componente Transform de un *GameObject* en toda la red.

Figura 12. *Network Transform*



NetworkManagerHUD

El componente *NetworkManagerHUD* proporciona una interfaz de usuario simple para gestionar la partida en red.

Reto 1

Añadid un modo temporal de arrancar fácilmente el juego en modo servidor y en modo cliente a partir de la escena *Main*.

Concretamente, la propiedad ***Network Sync Interval*** del *NetworkTransform* permite ajustar la ratio de fluidez en la que se sincronizará la localización de este *GameObject* en concreto, especificando cada cuando se sincroniza (en segundos).

Reto 2

Realizad una serie de pruebas con diferentes configuraciones de *NetworkTransform* y analizad los resultados obtenidos.

6.5. Identificación del jugador

Inicialmente, todos *los GameObjects* que representan a los jugadores presentan las mismas características y resultan idénticos a simple vista. Por lo tanto, para que el jugador local pueda identificar cuál es su tanque, necesitamos realizar algún cambio que sea fácilmente visible en pantalla.

La función *OnStartLocalPlayer* es un buen lugar para hacer la inicialización del jugador local como, por ejemplo, la configuración de las cámaras de la configuración y los controles de entrada. Esta función solo es llamada por el *LocalPlayer* en su cliente. En este caso, vemos en la implementación la función *OnStartLocalPlayer*:

```
public override void OnStartLocalPlayer() {
     foreach (MeshRenderer child in GetComponentsInChildren<MeshRenderer>())    {
          child.material.color = Color.blue;
     }
 }
```

Esto hará que cada jugador pinte los materiales del *prefab* Player de color azul y que cada jugador vea de este color el tanque que controla en su instancia del juego.

Figura 13. Resultado de la aplicación del código

6.6. Disparo de proyectiles

Una característica común en los juegos multijugador es que las acciones de cada jugador tengan efecto sobre el resto de los jugadores. En este caso, se trata del disparo de proyectiles con el objetivo de derrotar al adversario. Para ello, será necesario revisar *TankController* para que llame a una función que se encargue de realizar esta acción remota a través de la red.

- Añadir una referencia al *prefab* del proyectil y otra al lugar donde se realizará el *spawn*.

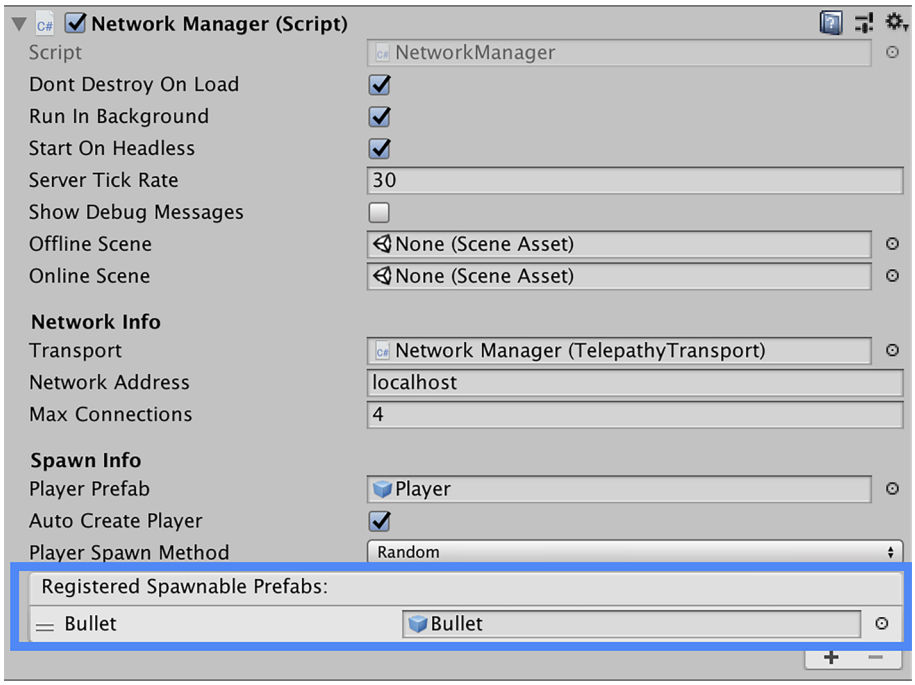
```
public GameObject bulletPrefab;
public Transform bulletSpawn;
```

- En la función *Update()*, añadir controles de entrada para la acción de disparo.

```
if (Input.GetKeyDown (KeyCode.Space)) {
   CmdFire();
}
```

Otros factores importantes a tener en cuenta que ya hemos visto en apartados anteriores son que el *prefab* del proyectil necesita un ***NetworkIdentity*** para identificarlo como único en la red, un ***NetworkTransform*** para sincronizar la posición y la rotación del proyectil, y que deberá ser registrado en el ***NetworkManager*** como *prefab* ***spawnable***.

Figura 14. Registro del *Bullet Prefab Spawnable*



De todas formas, dado que el proyectil no cambia de dirección o velocidad después de que haya sido disparado, no es necesario enviar actualizaciones de los movimientos. Cada cliente puede calcular de manera fiable la posición del proyectil una vez sabe su posición inicial. Es por ello que, una optimización sencilla, es simplemente no añadirle el componente *NetworkTransform*, de modo que la posición no se sincronice a través de la red y así se permita a cada cliente calcular la posición del proyectil por sí mismo, reduciendo el tráfico global de la red y mejorando rendimiento del juego.

A continuación, el *script TankController* también debe estar preparado para trabajar en red con la función *CmdFire*. Pero primero vamos a recapitular los pasos anteriores, ya que como hemos visto, el principio básico es que el servidor y todos los clientes están ejecutando el mismo código en los mismos *GameObjects* al mismo tiempo. Además, hemos visto una forma de control de estas ejecuciones mediante la propiedad *isLocalPlayer*.

Pero tenemos otra forma de control más potente llamada *Command*, que es llamada por el cliente pero ejecutada en el servidor. Para utilizar esta nueva llamada remota, debemos marcar la función con el atributo [*Command*]. Cualquier argumento en la función se envía automáticamente al servidor en la misma llamada como parámetro remoto.

Commands

Los *Commands* solo se pueden enviar desde el objeto del jugador local. Al declarar una función como *Command*, su nombre debe comenzar con el prefijo *Cmd*.

Siguiendo estas indicaciones, en el *scriptTaskController*, tenemos el atributo [*Command*] para la función *CmdFire*:

```
        [Command]
        void CmdFire()
```

Otro concepto único en este escenario es el concepto de *Network Spawning*, donde el mecanismo de *spawn* implica **algo más que generar una instancia**. Concretamente, significa crear un *GameObject* en el servidor y en todos los clientes conectados al servidor. El *GameObject* entonces será gestionado por el sistema de *spawn* y sus actualizaciones de estado se envían a los clientes cuando cambia en el servidor. Cuando su referencia sea destruida en el servidor, los *GameObjects* también serán destruidos en los clientes.

Ved también

Para más información sobre el atributo [*Command*], revisad la documentación oficial.

Para crear un objeto mediante este sistema, se utiliza la función *NetworkServer.Spawn(..)*. Por ejemplo, para crear el proyectil se haría de la siguiente forma:

```
NetworkServer.Spawn (bullet);
```

Finalmente, el resultado del *TankController* con estos conceptos incorporados es:

```
[Command]
void CmdFire() {

    var bullet = (GameObject) Instantiate(
        bulletPrefab,
        bulletSpawn.position,
        bulletSpawn.rotation);

    NetworkServer.Spawn (bullet);

    Destroy (bullet, 2.0f);
}
```

El primer paso para provocar que un proyectil afecte a su objetivo es añadir lógica de control de colisiones. Para este ejemplo, simplemente se destruye el *GameObject* del proyectil cuando este golpea cualquier otro objeto con un *Collider* asociado.

```
using UnityEngine;

public class Bullet : MonoBehaviour {

    void Start() {
        GetComponent<Rigidbody>().velocity = transform.forward * 6;
    }

    void OnCollisionEnter() {
        Destroy (gameObject);
    }
```

```
}
```

Vemos que el proyectil se instanciará en la posición correspondiente y a partir de aquí le definimos la velocidad deseada. Debido a que este proyectil está siendo gestionado por el *NetworkManager*, cuando se destruye el proyectil en el servidor, también será automáticamente destruido en todos los clientes, quedando todas las instancias consistentes.

Reto 3

Añadid otros tanques que no estén controlados por ningún jugador y que se vean afectados por los proyectiles que disparan los jugadores.

6.7. Control de la salud del tanque

Dentro de la sincronización de los estados de los jugadores, sabemos que los proyectiles hacen daño al impactar, afectando a la propiedad que representa la salud de los tanques. Por lo tanto, también se debe sincronizar el estado de salud de cada tanque en todas las instancias del juego.

Primero veamos cómo se gestiona la salud del jugador, a través del *script Health*, el cual gestiona el valor de vida actual y tiene declarado el valor máximo de este:

```
public const int maxHealth = 100;
public int currentHealth = maxHealth;
```

La lógica de control del daño recibido y la consecuente reducción de salud o muerte del jugador después de un impacto determinado es la siguiente:

```
public void TakeDamage (int amount) {

    currentHealth -= amount;
    if (currentHealth <= 0) {
        currentHealth = 0;
    }
}
```

Dado que los impactos en este juego solo son producidos por los proyectiles, debemos tener en consideración también el *script Bullet*, que es quien llama a la función *TakeDamage* durante la ejecución de la función *OnCollisionEnter*, que ya hemos visto y que una vez completada queda de la siguiente forma:

```
void OnCollisionEnter (Collision collision) {
```

```
        GameObject hit = collision.gameObject;
        Health health = hit.GetComponent<Health>();
        if (health != null) {
            health.TakeDamage (10);
        }

        Destroy(gameObject);
    }
```

En este proyecto proporcionamos un *GameObject* para la barra de vida (*healthbar*) con su componente de *script* (*Health*), de forma que permite visualizar la vida actual de los tanques como ocurre en muchos juegos, como por ejemplo en la saga *Command & Conquer*.

La estrategia que vamos a seguir en este caso es la ***authoritative server***, donde los cambios en la salud del jugador son controlados por la instancia del servidor. Estos cambios son posteriormente sincronizados por el cliente. Para aplicarla, debemos usar el mecanismo de sincronización de estado que nos proporciona la API *Mirror*.

Este mecanismo se habilita haciendo uso de otro atributo, concretamente la [*SyncVar*], que convierte las propiedades de los *GameObjects* en atributos de tipo *SyncVars*. A continuación, vamos a detallar cómo queda el *script* de *Health* con este mecanismo configurado.

Debemos usar el *namespace* de *UnityEngine.Networking*, extender la clase *NetworkBehaviour* y configurar la propiedad *currentHealth* como *SyncVar*:

```
        using Mirror;

        public class Health : NetworkBehaviour {

            [SyncVar]
            public int currentHealth = maxHealth;
```

También debemos añadir una comprobación en la función *TakeDamage* para que solo el servidor sea quien aplica el cambio de daño.

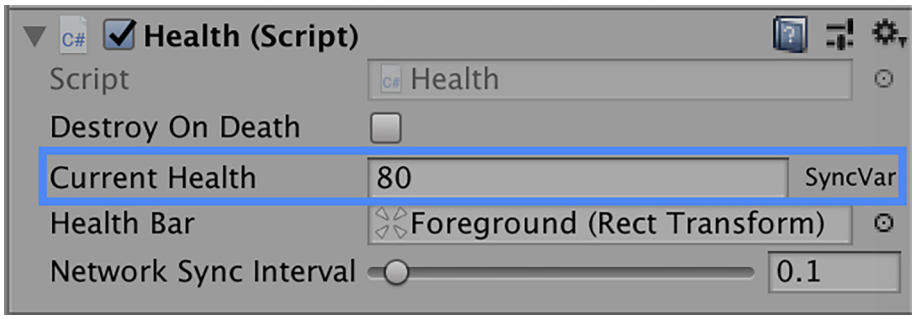
```
        if (!isServer)  {
            return;
        }
```

Así es como la salud del jugador solo se modifica en el servidor y es posteriormente sincronizada en todos los clientes.

Sin embargo, en este punto del proyecto, el *GameObject* de *healthbar* no funciona correctamente en todas las instancias del juego, sin visualizarse en los clientes como debería.

Si nos centramos en la variable *currentHealth*, esta es pública y se puede ver en el editor. Si el editor se está ejecutando en modo cliente, no como servidor, utilizando el inspector somos capaces de ver la salud actual del jugador.

Figura 15. *Current Health*



También podemos ver funcionando las barras de salud en la escena, pero solo en local, en el cliente actual y no en los otros clientes, como debería ser. Esto se debe a que no estamos sincronizando el valor de *RectTransform* a través de la red ni ejecutando el código para ajustar el tamaño de la *healthbar* en los otros clientes, ejecutándose solamente en el cliente local y en el servidor.

Healthbar en local

La razón por la que *healthbar* solo trabaja en el *host* cliente se debe a que es local en el servidor, compartiendo la misma escena.

Por lo tanto, el siguiente paso es sincronizar el *RectTransform* de la *healthbar*. Para ello, vamos a hacer uso de otra herramienta de sincronización de estado: el *SyncVar hook*. Los *SyncVar hooks* enlazan con una función *SyncVar*. Estas funciones se invocan en el servidor y todos los clientes son actualizados cuando cambia el valor del *SyncVar*.

Ved también

Para más información sobre los mecanismos de *SyncVars* podéis revisar la documentación oficial sobre sincronización del estado de *Mirror*.

Para empezar, los cambios en la *healthbar* se producen en la función *OnChangeHealth(..)*.

```
private void OnChangeHealth (int newHealth) {
    healthBar.sizeDelta = new Vector2 (currentHealth, healthBar.sizeDelta.y);
}
```

Vale la pena señalar que esta función debe tener un parámetro del mismo tipo que la variable con el atributo [*SyncVar*], en este caso *currentHealth*, y que el valor actual de la *SyncVar* se enviará a la función con *SyncVar hook* como parámetro adjunto. Para establecer un enlace a esta nueva función en el atributo *SyncVar* para *currentHealth*, debemos declararlo de la siguiente forma:

```
[SyncVar(hook = "OnChangeHealth")]
public int currentHealth = maxHealth;
```

Ahora sí, cuando el valor de la propiedad *currentHealth* cambie, *OnChangedHealth(..)* será llamado en el servidor y en todos los clientes para actualizar la *health bar*. Y así es como conseguimos que la *health bar* sea sincronizada en todas las instancias del juego. Por lo tanto, cuando los jugadores se disparan entre sí, todas las barras de salud deben reflejar el valor de la salud actual del jugador.

Reto 4

¿Qué mecanismos debemos utilizar para implementar la funcionalidad del cargador? Pensad en solucionar tanto la gestión del número de proyectiles como la forma de mostrar esta información.

6.8. Muerte y reaparición

Hasta este punto, nada ocurre si la salud del jugador llega a cero, excepto un mensaje en la consola del servidor. Para finalizar el *gameplay* de este juego, debemos revisar las acciones de muerte y *respawn* del jugador, es decir, cuando la salud llega a cero, este debe desaparecer y a continuación reaparecer en otro punto del mapa con la salud restaurada.

Este último paso nos permite introducir otro mecanismo para la sincronización de estado: las llamadas ***ClientRpc***. Estas llamadas remotas permiten enviar desde cualquier *GameObject* generado en el servidor con un componente *NetworkIdentity*.

Nota

A pesar de que una función *ClientRpc* se llama en el servidor, en realidad será ejecutada en los clientes.

Por lo tanto, podemos afirmar que las llamadas *ClientRpc* son el mecanismo inverso a un *Command*, ya que como hemos visto anteriormente, estos son llamados en el cliente, pero son ejecutados en el servidor. Además, de forma similar a Command, para convertir una función en una llamada *ClientRpc*, necesitamos añadir el atributo [*ClientRpc*] y el prefijo *Rpc* al nombre de la función. Además, cualquier argumento será automáticamente pasado como parámetro a los clientes como parte de la llamada *ClientRpc*.

Ved también

Para más información sobre las llamadas *ClientRpc*, podéis revisar la documentación oficial de *Mirror*.

Por lo tanto, para habilitar el mecanismo de *respawning* en el juego necesitamos la función *Respawn* con prefijo *Rpc* y el atributo [*ClientRpc*] dentro del *script Health*.

```
[ClientRpc]
void RpcRespawn() {
    if (isLocalPlayer) {
        transform.position = spawnPoint;
    }
}
```

La llamada se realiza en la función *TakeDamage* del servidor, cuando la salud actual del jugador llega a cero. Además, el código de *TakeDamage* debe restablecer la salud actual del jugador al máximo.

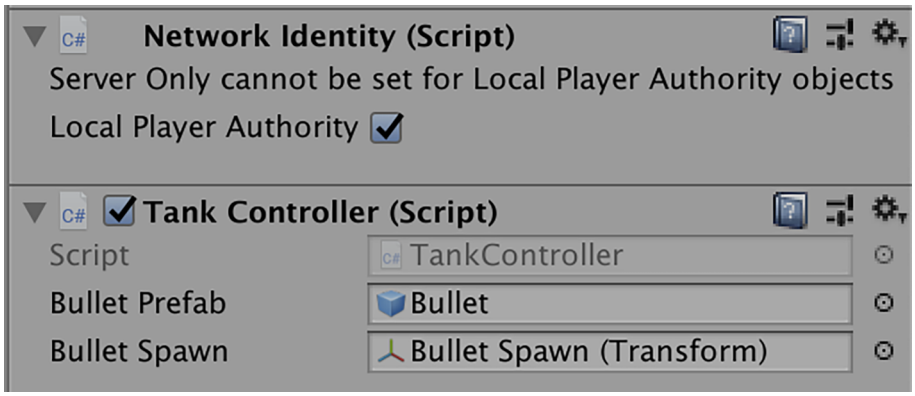
```
        currentHealth = maxHealth;
        RpcRespawn();
```

Hasta ahora, el cliente controla la posición del jugador *GameObject* local, dado que tiene activada la propiedad **Local Player Authority** en el *NetworkIdentity* del *prefab* del tanque del jugador.

Nota

Recordad que *RpcRespawn()* será llamada en el servidor, pero invocada en los clientes.

Figura 16. Propiedad *Local Player Authority*



Es por ello que, al llegar la vida a cero, el servidor simplemente restaura la posición del *GameObject* del tanque al origen, el cliente tendría la autoridad de detener esta acción.

Para evitar que esto ocurra, el servidor procede de forma que explícitamente le indica al cliente mover el tanque del jugador a la posición de reinicio mediante una llamada *ClientRpc*. De esta forma, esta posición se sincroniza a través de todos los clientes gracias al componente *NetworkTransform* del *GameObject* del jugador.

Podéis comprobar que funciona correctamente cuando un jugador dispara a otro y le hace suficiente daño como para dejar su salud a cero; entonces el *GameObject* del tanque dañado desaparece, vuelve al origen y su salud actual es restaurada al máximo.

Reto 5

Destruid los enemigos NPC al quedarse con la salud a cero después de ser alcanzados por proyectiles de los jugadores.

6.9. *Lobby*

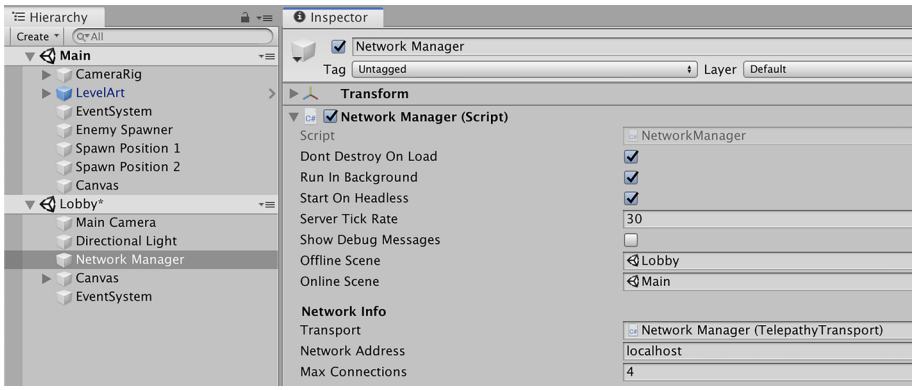
Finalmente, para completar el proyecto, vamos a hablar de la escena *Lobby* que nos permite crear una partida como *host* o unirnos a otras partidas ya creadas.

Figura 17. Escena *Lobby*

Para ello, el primer paso es crear la escena *Lobby* y mover aquí el *GameObject Network Manager*. Dado que vamos a crear un menú con a interfaz de usuario de Unity, vamos a requerir también el *GameObject* de tipo Event System para recoger las interacciones del usuario.

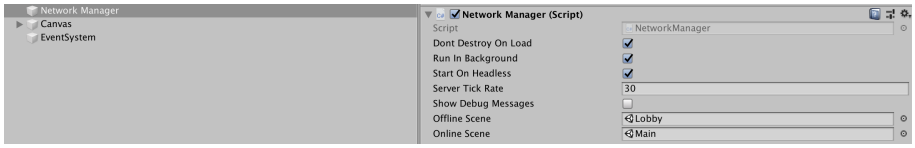
Cargando las dos escenas al mismo tiempo, nos resulta muy sencillo mover o copiar *GameObjects* de una a otra.

Figura 18. El *Network Manager* de la escena *Lobby*



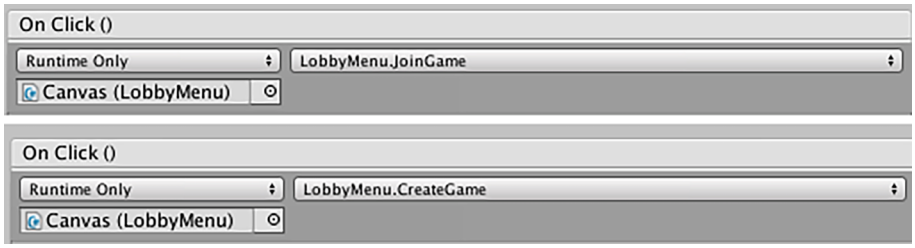
Con la escena *Lobby* y el *script LobbyMenu* veremos implementada la forma más simple para arrancar la partida multijugador en red local como *host*. Primero le indicamos al *NetworkManager* que cuando no esté conectado enseñe la escena *Lobby* y cuando lo esté enseñe la escena *Main*.

Figura 19. Configuración de escenas *Lobby-Main* en el *Network Manager*



A continuación, necesitamos la interfaz gráfica para interactuar; a partir del *canvas*, añadimos dos botones similares que nos permitirán llamar a las acciones para crear una partida y otra para unirse.

Figura 20. Configuración de botones de llamada a la acción



Estas funciones requieren de las funciones de red necesarias tanto para gestionar la partida (*NetworkManager*) especificando la IP del servidor para que los clientes puedan conectarse a él. Como vemos, *LobbyMenu* se implementa de la siguiente forma:

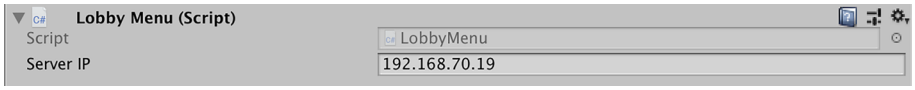
```
using UnityEngine;
using Mirror;

public class LobbyMenu : MonoBehaviour {
    private NetworkManager manager;
    public string serverIP = "localhost";

    void Awake(){
        manager = FindObjectOfType<NetworkManager>();
    }

    private void CreateGame() {
        if (!NetworkClient.isConnected && !NetworkServer.active) {
            if (!NetworkClient.active) {
                manager.StartHost();
            }
        }
    }
```

```
    private void JoinGame() {
        if (!NetworkClient.isConnected && !NetworkServer.active) {
            if (!NetworkClient.active) {
                manager.networkAddress = serverIP;
                manager.StartClient();
            }
        }
    }
}
```

En las versiones de la HLAPI de UNet disponíamos de funcionalidades para hacer Discovery de la red ejecutando un broadcast de forma totalmente abstraída y transparente para nosotros. *Mirror* no proporciona ninguna funcionalidad de este tipo y esto nos obliga a tener configurada de forma predeterminada la IP del servidor que hospedará las partidas.

En cualquier caso, mostramos este código a efectos demostrativos. Obviamente, la IP del servidor tendría que ser configurable si queremos permitir el juego en LAN o bien utilizando un servicio de *matchmaking*, como veremos en el siguiente módulo.

Figura 21. *Lobby Menu*



En el *script Lobby Menu* configuramos la IP del servidor si queremos probar nuestro juego en LAN, o bien *localhost* si queremos probar nuestro juego localmente.

En el punto del proyecto en red, el servidor y todos los clientes se ejecutan en un escenario donde el servidor hace al mismo tiempo de jugador y de *host*. Si quisiéramos un escenario con un servidor dedicado exclusivamente, deberíamos tener tres instancias del juego para dos jugadores: una de servidor y dos clientes, con dos *GameObjects* para los jugadores en cada una, para un total de seis *GameObjects* de jugador diferentes.

Reto 6

Modificad el proyecto para que en el *Lobby* aparezca la opción alternativa de iniciar una sesión de servidor dedicado.

Ya solo queda, aunque fuera del alcance de este proyecto, mantener y administrar estas conexiones, así como las tareas de revisar las desconexiones y las reconexiones de los jugadores y el servidor, y que el juego sea capaz de ser jugado continuamente.

Reto 7

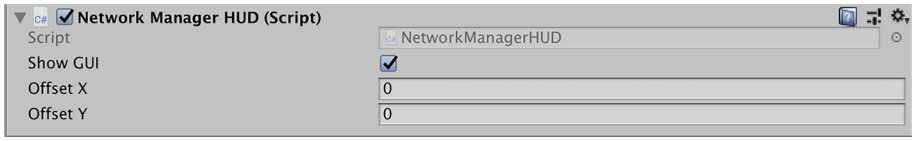
Mostrad por consola los datos de red más relevantes del servidor y de los clientes conectados.

6.10. Solución a los retos propuestos

Reto 1

El componente *NetworkManagerHUD* funciona con *NetworkManager* y proporciona una interfaz de usuario sencilla para controlar el estado de la red del juego cuando este se ejecuta.

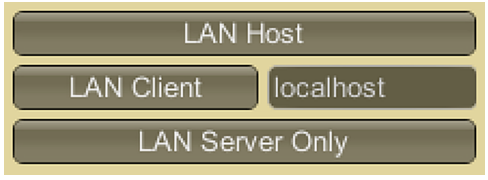
Figura 22. *Network Manager* HUD



Para probar el movimiento en modo multijugador debemos tener dos instancias del juego ejecutándose simultáneamente, una vez hayamos generado el juego para ejecutarse en nuestra máquina.

La primera instancia de juego que arranquemos mostrará la interfaz gráfica del *NetworkManagerHUD* que se muestra de la siguiente forma.

Figura 23. Interfaz gráfica del *Network Manager* HUD



Para probar el multijugador:

- En esta primera instancia, debemos hacer clic en *LAN Host* para crear la sesión de juego y deberíamos empezar a jugar en solitario esperando al siguiente jugador.

- Arrancamos una segunda instancia y hacemos clic esta vez en *LAN Client* para conectar al *host* de la primera instancia.

- De este modo, deberíamos ver en ambas instancias a los jugadores conectados y las dos pantallas deberían mostrar la misma escena.

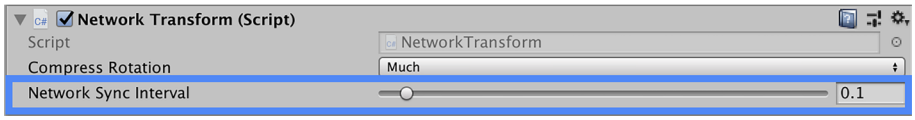
- Teniendo la ventana de una de las instancias activa y usando los controles del jugador, debería moverse en ambas pantallas.

Reto 2

En este punto del proyecto, los dos *GameObjects* del jugador deben ser completamente independientes entre sí y estar sincronizados con ambas instancias del juego, de forma simultánea. Pero podemos notar que, según la configuración del *Network Sync Interval* que tengamos cuando se ejecutan dos instancias del proyecto, el jugador remoto no parece moverse suavemente en el cliente local, dando una sensación de **movimiento a saltos**.

Hay que considerar en este punto la velocidad en la que los datos se pueden mover entre los clientes y el servidor. Por ello, *NetworkTransform* tiene un parámetro de configuración que justamente nos permite establecer la frecuencia con la que el objeto *NetworkTransform* envía datos de sincronización por red.

Figura 24. *Network Transform*



La frecuencia de las actualizaciones a través de una red entre los clientes y el servidor puede tener un gran impacto sobre cómo percibe el juego el jugador. En este punto, vale la pena tener en cuenta algunos puntos clave. Habrá que establecer un equilibrio entre la frecuencia con la que ocurre la sincronización de estado y el rendimiento del juego. Si un juego trata de sincronizar demasiados datos con demasiada frecuencia por red, el rendimiento del juego sufrirá efectos contraproducentes. Si un juego no sincroniza los datos con suficiente frecuencia, entonces la sensación del juego es poco fluida.

Hay que tener en cuenta que en todos los juegos multijugador por red nunca se consigue una sincronización perfecta, ya que esto es físicamente imposible. Dos o más clientes remotos no pueden estar en el mismo estado al mismo tiempo debido simplemente al tiempo que se tarda en transferir los datos entre ellos. Los jugadores, sin embargo, necesitan sentir como si su instancia estuviera en sincronización para tener la mejor experiencia, a pesar de que en realidad las instancias son ligeramente diferentes en su estado exacto.

Reto 3

NPC

A los personajes no controlados por el jugador se los conoce como NPC.

Nuestro proyecto actual se ha centrado principalmente en el jugador. La mayoría de los juegos, sin embargo, tienen muchos *GameObjects* que no son controlados por el jugador (por ejemplo, NPC, *Non-Player Character*) u otros *GameObjects* de tipo ambiental, pero que también coexisten en el mundo del juego.

En este reto, nos centraremos en los *GameObjects* que consideramos directamente enemigos. Mientras que del *GameObject* del jugador se hace *spawn* cuando un cliente se conecta al *host*, los *GameObjects* enemigos necesitan ser controlados por el servidor.

Así pues, el primer paso es crear un *GameObject* llamado *Enemy Spawner* que se encargue de crear los enemigos mediante un componente *NetworkIdentity*, configurado como *Server Only*. Al activar esta propiedad, evitaremos que el *Enemy Spawner* se envíe a los clientes. Para complementar este *GameObject*, necesitaremos definir su comportamiento mediante un componente de *script* que hemos llamado *EnemySpawner*:

```
using UnityEngine;
using Mirror;

public class EnemySpawner : NetworkBehaviour {
    public GameObject enemyPrefab;
    public int numberOfEnemies;

    public override void OnStartServer() {
        for (int i = 0; i < numberOfEnemies; i++) {
            var spawnPosition = new Vector3(
                Random.Range(-8.0f, 8.0f),
                0.0f,
                Random.Range(-8.0f, 8.0f));

            var spawnRotation = Quaternion.Euler(
                0.0f,
                Random.Range(0, 180),
                0.0f);

            var enemy = (GameObject) Instantiate (enemyPrefab, spawnPosition,
                                spawnRotation);
            NetworkServer.Spawn (enemy);
        }
    }
}
```

Este *script* hereda de *NetworkBehaviour*, y sobrescribe la función *OnStartServer()*. De esta forma, cuando el servidor arranca, genera un número de enemigos con posiciones y rotaciones aleatorias y finalmente ejecuta el mecanismo de *spawn* mediante la función *NetworkServer.Spawn (enemy)*. Notad que la función *OnStartServer()* es muy similar a la que ya hemos visto antes: *OnStartLocalPlayer()*. En este caso, *OnStartServer()* es llamado en el servidor cuando este empieza a escuchar de la red.

Por otro lado, para facilitar esta tarea de *spawn* de enemigos, vamos a usar el *prefab* Player como base para el *GameObject* del tanque enemigo. Este *prefab* también necesita los componentes *NetworkIdentity* y *NetworkTransform* ya incluidos, así como el *script Health* y la configuración de la *healthbar*. En cambio, debemos eliminar el *GameObject* hijo de *Bullet Spawn* y el componente de *Tank Controller*.

Ahora debemos registrar este *prefab* en el *Enemy Spawner* para que sea utilizado para la creación de enemigos.

Figura 25. *Enemy Spawner*

Notad que al no cambiar el color del *prefab*, tanto los jugadores enemigos como controlados por otros jugadores remotos, como los enemigos *NPC*, tendrán el mismo color base. Además, debido a que el *prefab* Enemy también tiene los *scripts Health* y *Bullet*, detectará a los enemigos de la misma forma que si fueran del tipo Player.

Reto 4

Para implementar la gestión del número de proyectiles, seguiremos la misma idea que en el caso de la salud del jugador, una propiedad con el número máximo de proyectiles del cargador y otra con el valor actual utilizando el mecanismo de *SyncVar*.

A la hora de mostrar esta información, podríamos mantenerla solo en local, y de hecho es muy habitual en este tipo de juegos que solo el jugador vea esta información en su GUI. Si quisiéramos darle un toque estratégico y que los otros jugadores también supieran el estado del cargador, deberíamos seguir el mismo ejemplo que la barra de vida y utilizar el mecanismo de *SyncVar hook*.

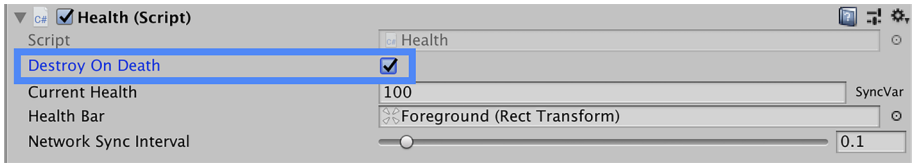
Reto 5

Si ejecutamos el proyecto en este punto, veremos que se crean cuatro enemigos no controlados al crear una nueva sesión de juego, que no pueden ser destruidos mediante disparos de proyectiles por nuestro tanque.

Para ello debemos ajustar el *script Health* para que tenga en cuenta la diferencia entre los tanques controlados y los no controlados. La forma más sencilla es crear una nueva propiedad en este *script*:

```
        public bool destroyOnDeath;
```

Figura 26. *Script Health*



En la función *TakeDamage()*, debemos comprobar esta propiedad cuando la vida actual llega a cero antes de realizar el *respawn*:

```
        if (currentHealth <= 0) {
            if (destroyOnDeath) {
                Destroy(gameObject);
            }
            else {
                currentHealth = maxHealth;
                RpcRespawn();
            }
        }
```

Ahora, la acción de disparar a los enemigos hará que pierdan la salud y cuando esta llegue a cero los enemigos serán destruidos. Los jugadores, sin embargo, continúan teniendo el comportamiento anterior donde se mueven de nuevo al punto de origen al morir con su salud restaurada al máximo.

Reto 6

Para poder implementar un servidor dedicado con esta solución, deberíamos primero permitir a la escena *Main* ejecutarse en modo servidor y no en modo *host*, como hacíamos hasta ahora desde el *Lobby*.

Para ello, debemos añadir una nueva función, que llamaremos *RunServer()*, muy similar a *CreateGame()*, en el *script* de *LobbyMenu*:

```
    private void RunServer() {
        if (!NetworkClient.isConnected && !NetworkServer.active) {
            if (!NetworkClient.active) {
                manager.StartServer();
```

```
            }
        }
    }
```

Una vez tenemos esta propiedad, debemos modificar la interfaz gráfica para añadir también un botón similar al *CreateButton* que llame a esta nueva función.

Cuando tengamos este nuevo mecanismo preparado, podemos generar el juego de nuevo y arrancar dos instancias al mismo tiempo para comprobar su funcionamiento.

Figura 27. Arranque del juego con dos instancias simultáneas

Reto 7

Como consejo típico en estos casos, las primeras pruebas es mejor empezar arrancando el servidor dedicado desde el editor de Unity y hacer *debug* viendo las trazas de *log* para comprobar que todo sucede como esperamos. Después podemos arrancar el primer cliente y posteriormente el segundo cliente para analizar lo que está pasando de forma directa.

Para obtener datos interesantes, vamos a añadir a las funciones del *LobbyMenu* que crean el *server*, *host* o cliente la llamada a la siguiente función:

```
    private void AddressData() {
        if (NetworkServer.active) {
            Debug.Log ("Server: active. IP: " + manager.networkAddress + " -
Transport: " + Transport.activeTransport);
        }
```

```
        else {
            Debug.Log ("Attempted to join server " + serverIP);
        }

        Debug.Log ("Local IP Address: " + GetLocalIPAddress());
    }
```

Para obtener la dirección IP de nuestra máquina nos ayudaremos del método auxiliar *GetLocalIPAddress()* que nos la devuelve:

```
    private static string GetLocalIPAddress() {
        var host = System.Net.Dns.GetHostEntry (System.Net.Dns.GetHostName());
        foreach (var ip in host.AddressList) {
            if (ip.AddressFamily == System.Net.Sockets.AddressFamily.InterNetwork) {
                return ip.ToString();
            }
        }

        throw new System.Exception("No network adapters with an IPv4 address in the system!");
    }
```

A partir de este punto, si necesitamos más control sobre los posibles eventos que suceden dentro de nuestro *NetworkManager*, como por ejemplo reaccionar cuando desde el servidor se detecta que un cliente se conecta (*OnServerConnect*) o desconecta (*OnServerDisconnect*), o bien cuando un cliente detecta que el servidor ha desaparecido (*OnClientDisconnect*), tenemos que sobrescribir nuestro propio *NetworkManager*. Si además requerimos de herramientas más sofisticadas entonces podemos analizar el tráfico generado por nuestro juego mediante *sniffers*, *profilers*, etc.

6.11. Sumario

Con este proyecto hemos cubierto la mayor parte de los conceptos básicos de la API de *Mirror*, que como hemos dicho está fuertemente basada en la obsoleta HLAPI de UNet, para hacer un simple juego multijugador en red.

Como hemos visto, cuando se utiliza *Mirror* el servidor y todos los clientes están ejecutando el mismo código al mismo tiempo en los *scripts* asociados a los *GameObjects*.

Hemos visto cómo podemos mantener las transformadas de nuestros *GameObjects* en red sincronizada con la del *NetworkIdentity* y el componente *NetworkTransform*. Hemos trabajado con otros mecanismos de sincronización de *SyncVars* y *SyncVar Hooks*, y ahora podemos mantener el valor de las variables de forma consistente a pesar de los cambios.

También hemos cubierto los mecanismos RPC disponibles en *Mirror*. Tanto *Command* como *ClientRpc*, siendo complementarias en lo referente al origen y destino entre cliente y servidor.

Por último, hemos diferenciado las escenas de juego y de *Lobby*, así como las estrategias de cliente y servidor (*host*) al mismo tiempo o clientes y servidor dedicado. Estas opciones las tendremos que tener muy en cuenta a la hora de distribuir y desplegar el juego.

Esperamos que este proyecto sea útil como punto de partida para comprender el uso de los mecanismos para videojuegos multijugador en red y como visión general de los aspectos más importantes de la API de alto nivel proporcionada por Unity.

Resumen

Como hemos visto, el multijugador en red requiere conocimientos y nociones muy avanzadas en temas de redes. No es el objetivo de este módulo de introducir aspectos avanzados en programación de red, pero sí tener muy claras las implicaciones de las topologías y los mecanismos escogidos, y las ventajas e inconvenientes que pueden tener a la hora de desarrollar nuestros proyectos.

Con la serialización y la compresión, se nos permite empaquetar un objeto y enviarlo a un *host* remoto, así como enmarcar estos datos de manera que el *host* remoto pueda crear o encontrar el objeto apropiado para recibir los datos.

La replicación es un mecanismo que requiere una estrategia adecuada para cada tipo de juego y, por lo tanto, se tienen que considerar sus diferentes variantes. En cambio, las invocaciones remotas vía RPC son muy recomendables para cualquier tipo de juego, ya que simplifican el protocolo de comunicación a implementar.

De esta forma, debemos conocer a fondo mecanismos de alto nivel, siendo la capa de aplicación donde trabajan los juegos y, por lo tanto, dejar la elección del protocolo de la capa de transporte para una segunda fase.

Además, debemos estudiar cómo optimizar y evitar problemas intrínsecos de los juegos multijugador, donde una de las soluciones clásicas sigue siendo centralizar la lógica del juego en una entidad neutral, típicamente llamada como servidor; esta es la opción tan conocida entre los jugadores habituales, los denominados *servidores dedicados*.

Finalmente, hemos visto como Unity, gracias a la API *Mirror*, nos permite desarrollar un videojuego en multijugador. También podemos entender que un videojuego en local, aunque sea multijugador, no puede ser migrado directamente a multijugador en red, sino que se requiere una modificación a fondo de este. Una práctica habitual consiste en realizar primero el multijugador de red y *a posteriori* el multijugador, o incluso el modo de un jugador.

Bibliografía

Alexandre, Thor (2005). *Massively Multiplayer Game Development 2 (Game Development)*. Rockland, MA: Charles River Media, Inc.

Armitage, Grenville; Claypool, Mark; Branch, Philip (2006). *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. Hoboken, NJ: John Wiley & Sons, Ltd.

Barron, Todd (2001). *Multiplayer Game Programming*. Boston, MA: Thomson Course Technology.

Beau Williamson (1999). *Developing IP Multicast Networks*. Indianapolis, IN: Cisco Press.

Tanenbaum, Andrew S.; Van Steen, Maarten (2006). *Distributed Systems: Principles and Paradigms* (2.ª ed.). Upper Saddle River, NJ: Prentice-Hall, Inc.

Yahyavi. Amir; Kemme, Bettina (2013). «Peer-to-peer architectures for massively multiplayer online games: A Surve». *ACM Comput. Surv.* (vol. 46, nº 1, art. 9).

Young, Vaughan (2005). *Programming a Multiplayer FPS In DirectX*. Hingham, MA: Charles River Media.