Jocs multijugador en xarxa

PID_00270022

Carles Pairot Gavaldà
Rubén Mondéjar Andreu

Temps mínim de dedicació recomanat: 6 hores

Universitat Oberta de Catalunya

Carles Pairot Gavaldà **Rubén Mondéjar Andreu**

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Javier Cánovas Izquierdo (2020)

Primera edició: febrer 2020

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

Índex

Introducció

En aquest mòdul es presenten els conceptes de xarxes de computadors que necessitarem per entendre com funcionen les eines per al desenvolupament de videojocs multijugador, mitjançant un breu repàs de les nocions sobre xarxes, tant locals com globals, així com una comparativa de les opcions disponibles.

Començarem analitzant resumidament la tecnologia que hi ha darrere d'Internet per saber com es fa la transferència d'informació entre dos dispositius connectats a la xarxa. Posteriorment, s'expliquen els mecanismes que es construeixen a sobre d'aquesta tecnologia per abstreure als desenvolupadors de les problemàtiques més habituals.

Seguidament, parlarem de com s'organitzen els dispositius en xarxa mitjançant una topologia per treballar de manera conjunta. Analitzarem breument els dos vessants més utilitzats i els avantatges i desavantatges de cadascun. A més, hem de conèixer els problemes més freqüents que haurem d'afrontar un cop el nostre joc comenci a funcionar en una xarxa global. Hi ha diferents tècniques per evitar aquests problemes i en veurem algunes.

Finalment, entrarem de ple en el desenvolupament de videojocs multijugador amb Unity, tot veient quines API tenim disponibles i en què consisteix cadascuna. Atesa la forma en què estan construïdes i es complementen entre si, la recomanació és començar el nostre desenvolupament utilitzant la capa d'alt nivell.

Objectius

En aquest mòdul didàctic es presenten a l'estudiant els coneixements necessaris per aconseguir els objectius següents:

1. Entendre les nocions generals i subjacents per treballar en entorns de xarxes.

2. Comprendre les diferents topologies de xarxa, així com les seves característiques principals.

3. Conèixer els conceptes bàsics, els problemes i les solucions en el desenvolupament de jocs en xarxa.

4. Ser capaç de programar un videojoc de multijugador en xarxa amb Unity.

1. Breu repàs a les xarxes i Internet

Si bé el concepte de joc multijugador no està vinculat exclusivament als jocs en línia, és evident que aquest és el sistema més utilitzat actualment per dur a terme aquest tipus de mecàniques. Tot seguit, s'ofereix una breu visió general del conjunt de protocols TCP/IP, i els protocols i els estàndards associats que participen en la comunicació per Internet, inclòs un breu repàs dels que són més rellevants per a la programació en xarxa.

1.1. Arquitectura d'Internet

La necessitat de comunicar diversos dispositius electrònics va néixer amb ells. Projectes com ARPAnet van crear les primeres xarxes d'ordinadors a petita escala, però el gran salt qualitatiu es va produir amb l'aparició d'Internet com a xarxa de comunicació global. L'èxit d'Internet rau en el fet que proporciona una infraestructura que permet que un gran nombre de dispositius completament diferents puguin interactuar entre ells i intercanviar serveis i informació.

La comunicació entre dos dispositius connectats en una xarxa es realitza mitjançant el que coneixem com a protocol de comunicació, és a dir, un conjunt de normes que defineixen la llengua en què dos ordinadors parlen entre ells. El conjunt de protocols avui més estès i que s'utilitza actualment a Internet es coneix com a model TCP/IP (*Transmission Control Protocol / Internet Protocol*).

1.2. Capes TCP/IP

El model TCP/IP es basa en una arquitectura de cinc capes que realitzen funcions específiques dins del procés de comunicació entre dues aplicacions, que són:

- **Capa física**. És la capa de nivell més baix, on la informació es codifica amb senyals elèctrics o ones de ràdio i es transmet entre dues màquines per un mitjà físic.

- **Capa d'accés al mitjà**. Cobreix el hardware i el protocol de transmissió. Cada implementació utilitza un estàndard de la indústria. Per exemple: ethernet, RDSI, coaxial, *wireless*, etc.

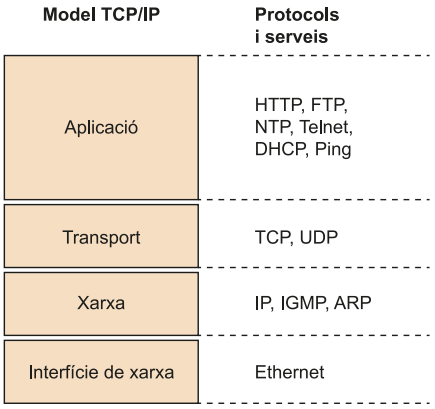
- **Capa de xarxa**. Responsable de la manera com es comuniquen diversos punts d'una xarxa. S'utilitza un protocol senzill de transmissió de paquets anomenat *Internet Protocol* o IP .

- **Capa de transport**. Responsable de la manera com es divideix i reordena un flux continu d'informació en paquets IP. També és responsable

d'assegurar l'absència d'errors. Hi ha dos protocols en aquesta capa: *Transport Control Protocol* (TCP) i *User Datagram Protocol* (UDP). Els analitzarem detalladament més endavant.

- **Capa d'aplicació**. En aquesta capa es defineixen els protocols de les aplicacions, per exemple: SMTP, POP3, IMAP per al correu electrònic, HTTP per a continguts web, etc. Per a una aplicació pròpia cal dissenyar el nostre propi protocol de comunicació.

Figura 1. Protocols i serveis en el model TCP/IP



Analitzem com es realitza una transmissió d'informació entre dos ordinadors utilitzant l'arquitectura TCP/IP:

- Quan una aplicació vol enviar dades o informació a una altra, la informació es divideix en paquets de dades. La mida d'aquests paquets està prefixada pel sistema operatiu. Normalment són bastant petits (uns 1.500 bytes per paquet).

Encapsulació

És un mètode de disseny modular de protocols de comunicació en el qual les funcions lògiques d'una xarxa són abstretes ocultant informació a les capes de nivell superior.

- Tots els paquets han de travessar totes les capes de l'arquitectura TCP/IP dins de l'ordinador. Cadascuna d'aquestes capes afegeix informació extra (**encapsulació**) perquè es pugui interpretar a quin ordinador va dirigit, i, un cop a la capa física, es realitza la comunicació real amb l'altre ordinador. Aquesta comunicació pot passar a través de nodes intermedis.

- Quan el destinatari rebi el paquet, aquest travessa l'arquitectura en sentit invers utilitzant la informació continguda en les dades addicionals fins que es lliura a l'aplicació corresponent.

- El destinatari ha de recollir tots els paquets enviats i muntar-los novament per poder recuperar la informació original.

1.2.1. Adreçament

Hem vist que en cadascuna de les capes s'afegeix informació extra que ens ajuda a controlar la transmissió dels paquets. Un dels elements més importants que s'afegeix a tots els nivells és l'adreça, que permet identificar a quina aplicació i a quina màquina volem enviar la informació. En el model TCP/IP hi ha una adreça per a cada nivell:

- **Capa d'accés al mitjà**. L'adreça s'identifica mitjançant el MAC (*Media Access Control*) de la targeta de xarxa que intervé en la comunicació. S'utilitza per saber a quina màquina de la nostra xarxa local hem d'enviar les dades perquè s'apropin al seu destí.

- **Capa de xarxa**. En aquesta capa s'afegeix l'adreça IP, que identifica la localització exacta de la destinació. L'adreça IP consisteix en un número de 32 bits, que normalment es representa amb quatre xifres decimals de 8 bits separades per punts.

- **Capa de transport**. L'adreça ens permet saber a quina aplicació s'ha de lliurar la informació. Aquesta adreça es representa amb un número de 16 bits que s'anomena *port*.

- **Capa d'aplicació**. No hi ha cap estàndard i és l'aplicació mateixa que guarda i manté a quin component s'han de distribuir les dades.

1.2.2. Tipus de comunicació

El protocol de comunicació pot ser orientat a la connexió, en el cas que es requereixi un canal persistent entre les dues instàncies a través de les quals transmetem la informació, o no orientat a la connexió si no es requereix aquest canal per transferir les dades.

La comunicació **orientada a la connexió** requereix l'establiment d'un canal de dades a través d'una connexió inicial entre les dues instàncies que intervenen en la comunicació. Al principi, les dues instàncies negocien entre si i creen una connexió dedicada per poder parlar l'una amb l'altra. Un cop s'estableix el canal, les dues instàncies envien i reben els paquets de dades a través d'ell. En l'arquitectura TCP/IP, aquest tipus de connexió es realitza mitjançant el protocol TCP. Alguns exemples de protocols d'aplicació que funcionen amb ell són HTTP, POP3 o FTP.

En canvi, la comunicació **no orientada a la connexió** permet enviar paquets entre dues instàncies sense establir cap mena de connexió inicial. En aquest cas, el protocol utilitzat és l'UDP i com a exemples d'aplicacions basades en aquest tipus de connexió tenim diversos sistemes de transferència de vídeo o àudio, DHCP, DNS, etc.

Comparem els dos protocols amb una mica més de detall.

Taula 1. Protocols TCP/UDP

TCP	UDP
Necessita establir un canal de comunicació.	No necessita cap mena de connexió prèvia per poder enviar paquets.
Garanteix que les dades s'han transferit íntegrament entre les dues instàncies.	No garanteix que es rebin les dades.
Els paquets es reben en el mateix ordre en què s'envien.	Tampoc no garanteix que es rebin en el mateix ordre en què s'han enviat.
Els paquets amb dades poden tenir una mida de bytes variable.	Els paquets (en aquest cas s'anomenen datagrames) de dades són de la mateixa longitud.
Si hi ha algun problema de transmissió, els paquets es reenvien, creant més trànsit a la xarxa.	No existeix el reenviament de paquets.
És un protocol lent, perquè ha d'assegurar-se que totes les dades es reben i que estan en ordre.	És un protocol ràpid, ja que no requereix comprovacions addicionals.

En el context dels videojocs, l'elecció del tipus de comunicació és molt important. Aquesta decisió depèn del tipus de joc que vulguem implementar.

Per exemple, en un joc d'estratègia on cada moviment és clau per al funcionament del sistema, és preferible una connexió TCP que ens garanteixi que s'executen totes les accions empreses per l'usuari. En canvi, per exemple, en un *First Person Shooter* (FPS), en el qual es pot actualitzar el sistema moltes vegades per segon, no podem utilitzar un protocol TCP perquè podria alentir el sistema, així que possiblement la millor opció és utilitzar UDP i simplement no preocupar-nos pels paquets que es perden.

Una altra solució que podríem adoptar seria utilitzar una solució mixta, és a dir, podríem tenir al mateix temps un canal de control a través del qual es transmeti la informació clau amb una connexió TCP, i un canal de dades pel qual es transmeti la resta de la informació no crucial mitjançant el protocol UDP.

Un exemple d'aquesta idea seria enviar per TCP les dades relatives a aquells altres usuaris que són a prop i amb els quals podem arribar a interactuar, i d'altra banda, enviar per UDP les dades d'usuaris llunyans, amb els quals la possibilitat d'interactuar és gairebé nul·la.

Malgrat això, **cal evitar la temptació** d'aquest tipus d'aproximació híbrida a causa d'un problema col·lateral que comporta aquesta solució. El fet que ambdós protocols descansin sobre el mateix protocol IP pot generar problemes, com la congestió i la latència que indirectament aporta el protocol TCP al trànsit de paquets UDP quan s'utilitza al mateix temps.

Per això, una alternativa a TCP molt utilitzada en el món dels videojocs en xarxa és l'R-UDP (*Reliable* UDP). Aquest protocol és molt similar a UDP, però a través d'una implementació lleugera aconsegueix tenir la mateixa fiabilitat que TCP, sense l'*overhead* extra i els problemes esmentats.

UDP en Unity

Unity proporciona i utilitza diferents modalitats UDP, inclòs l'R-UDP.

2. Programació en xarxa

Control de les dades

La programació es pot fer de diferents maneres, depenent del control que vulguem tenir sobre el que s'envia i es rep.

La programació d'un joc en xarxa requereix dos elements imprescindibles. En primer lloc, hem de crear les connexions entre els clients, així com saber com enviar i gestionar la informació. I, en segon lloc, necessitem dissenyar o incorporar tècniques que ens permetin abstreure'ns de la comunicació entre les diferents instàncies remotes d'un mateix joc. És a dir, hem de dissenyar o utilitzar diferents mecanismes que ens permetin enviar i replicar l'estat del nostre joc i dels elements que el componen.

La primera opció és utilitzar les eines de baix nivell com els anomenats *sockets*, els quals ens proporcionen un accés directe a la capa de xarxa. En aquest sentit, tenim un control total de cada byte d'informació que es transmet a través de la xarxa, però això també implica que ens hem d'encarregar de qualsevol problema que pugui aparèixer, així com tractar amb múltiples connexions que requereixin estructures més complexes.

Una altra opció és utilitzar un mecanisme **d'alt nivell**, que gestioni mitjançant abstraccions totes les connexions concurrents i l'estat de les mateixes. En aquest cas, tenim menys control, però si aquesta capa superior s'implementa de manera eficient, reduirem considerablement la feina necessària per poder implementar les connexions, i ens podrem centrar en el disseny del joc.

Ambdues opcions són perfectament vàlides i complementàries a l'hora d'implementar un videojoc.

Seguint aquest ordre, primer veurem la programació bàsica en xarxa i després parlarem dels mecanismes que podem aplicar segons els requeriments que tinguem.

2.1. *Sockets*

L'eina bàsica per programar en xarxa es basa en el concepte de *socket*.

Un *socket* és una entitat que permet que un ordinador intercanviï dades amb altres ordinadors. Un *socket* identifica una connexió particular entre dos ordinadors i sempre es compon de cinc elements: una adreça de xarxa local (identificador IP), un port de la màquina local, una adreça de xarxa de destí, un port de la màquines de destí i el tipus de protocol (TCP o UDP).

El concepte de *socket* es va desenvolupar a la Universitat de Berkeley quan van començar a comunicar diversos ordinadors UNIX. A partir d'aquesta experiència, es va desenvolupar una biblioteca senzilla que permetia aprofitar aquest treball i escriure nous programes capaços de comunicar-se entre ells.

2.1.1. Ús de *sockets* TCP

Atès que TCP és un protocol que requereix connexió, primer cal establir-la entre els dos ordinadors abans de poder transmetre informació. A més, TCP ha de mantenir l'estat per poder reenviar els paquets perduts, i aquest estat ha de ser guardat d'alguna manera. Tot seguint l'API de Berkeley, els mateixos *sockets* emmagatzemen l'estat de la connexió. Això vol dir que un *host* necessita un *socket* únic per a cada connexió TCP que mantingui oberta.

En el nostre cas, parlarem de com implementar els exemples bàsics mitjançant la programació en C#, que podem incloure aplicant una adaptació simple dins dels nostres *scripts* en Unity. En qualsevol cas, els passos bàsics per establir la comunicació TCP serien:

Sockets en Unity

La biblioteca que ens permet utilitzar *sockets* en Unity és *System.Net.Sockets*.

- En primer lloc, hem d'importar les biblioteques de *sockets* per poder accedir a la interfície de xarxa.

- Un cop tinguem accés a la xarxa, obrim un nou *socket* pel qual enviarem i rebrem les dades. En aquest exemple comencem obrint un *socket* TCP, a l'adreça i el port del servidor, i d'aquesta manera obtenim el canal de *streaming*.

- Quan tenim el canal obert, podem enviar i rebre dades a través seu. En aquest cas, és un exemple de com enviar un text de prova, però com veurem, serà el protocol de nivell d'aplicació (en aquest exemple, HTTP) el que determina la manera com s'intercanviarà la informació.

Com a exemple, i per il·lustrar millor com funciona aquest protocol, vegem quin codi necessitem per poder realitzar un exemple simple. En primer lloc, veurem com instanciar i fer funcionar el servidor, que rep les connexions i envia la informació, i després veurem el client, que s'encarrega de connectar i rebre aquesta informació.

Per tant, començarem amb la classe ***TcpClient***, la qual permet tant rebre connexions durant el procés d'escolta, com a la inversa, sol·licitar les dades d'un recurs d'Internet mitjançant TCP. Les funcions i les propietats de *TcpClient* resumeixen els detalls per gestionar el *socket* i també per sol·licitar i rebre dades mitjançant TCP.

Per esperar una connexió TCP, és necessari establir un port TCP que el client ha de conèixer. *Internet Assigned Numbers Authority* (IANA) defineix els números de port per als serveis comuns com SSL o FTP. Els serveis fora de la llista

IANA poden ser ports dins de l'interval dels números 1.024 a 65.535. La imatge següent mostra com crear un servidor horari de xarxa mitjançant la classe ***TcpListener*** escoltant en el port 1755. Normalment hauríem d'utilitzar el port 13, però en ser inferior a 1.024, Unity ens denegarà l'accés ja que és un port reservat. En aquest exemple, quan s'accepta una sol·licitud de connexió entrant, el servidor horari respon amb la data i l'hora actuals de la màquina on s'executa:

```
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();
    }
}
```

Com es pot observar, *TcpListener* accepta les sol·licituds entrants amb la funció *AcceptTcpClient()* i per a cadascuna d'elles es crea un *socket* (*TcpClient*) que administra la connexió al client. Una particularitat en aquest codi és que s'ha hagut de crear un fil d'execució (*Thread*) separat per gestionar de manera concurrent les diferents connexions que aquest servidor pot rebre al mateix temps. Si no es fes així, el procés es penjaria fins a rebre una nova connexió. És per això que la funció *Start()* crea un nou *Thread* que executa en paral·lel la funció *NetworkCode()*. Anàlogament, la funció *OnDisable()* realitza la finalització neta del procés tot avortant el *Thread* actual i alliberant el port, ja que mentre s'està utilitzant, cap altre servidor el podrà utilitzar a la màquina on s'executa aquest procés .

Provant el codi

Per provar aquests codis d'exemple, n'hi ha prou d'afegir un *script* a un *GameObject* buit i executar el joc.

D'altra banda, perquè un client pugui establir una connexió TCP, a més del port on s'està oferint el servei, també requereix conèixer l'adreça del dispositiu de xarxa que allotja el servei. L'exemple següent mostra la configuració de *TcpClient* per connectar-se a un servidor en el port TCP 1755 i llegir-ne la informació de temps enviada pel 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. Ús de *sockets* UDP

Tot seguit, seguint el mateix esquema que en el cas de TCP, s'exposa com treballar a nivell client-servidor amb el protocol UDP.

Recordem que el **protocol de datagrames d'usuari** (UDP) és un protocol simple que ofereix un rendiment més alt per lliurar dades a un *host* remot. Tanmateix, el protocol UDP no està orientat a la connexió i per tant no garanteix l'enviament de datagrames UDP a l'extrem remot, ni que la seva recepció estigui en l'ordre en què van ser enviats.

A diferència de TCP, en aquest protocol l'esquema és més aviat productor-consumidor que client-servidor, ja que no hi ha connexió entre ells. Per poder enviar un datagrama mitjançant UDP, el productor ha de conèixer l'adreça de la xarxa del dispositiu que vol consumir aquesta informació, així com el número de port UDP que s'utilitza per comunicar-se.

A més, aquest protocol permet l'enviament de missatges a múltiples *hosts* en realitzar enviaments massius a grups, una acció coneguda com a *multicast* o indiscriminats, com a *broadcast,* segons el tipus de difusió, la xarxa implicada i la versió de protocol IP. És fàcil veure com aquest tipus de sistema de missatgeria és especialment útil si estem realitzant accions comunes en múltiples màquines, com els jocs multijugador en xarxa.

Vegem, d'exemple, com podríem implementar enviaments de tipus *broadcast*. Les difusions es poden dirigir a parts específiques d'una xarxa establint tots els bits d'identificador de *host*. Per exemple, per enviar una difusió per a tots els *hosts* d'una LAN identificada per 192.168.70.x, cal d'utilitzar l'adreça 192.168.70.255. Per utilitzar aquesta modalitat s'han de fer servir adreces de xarxa especials en xarxes basades en IPv4 mitjançant una màscara de xarxa concreta (255.255.255.0).

Vegeu també

Per a més informació sobre l'API de C# i *sockets,* podeu revisar la documentació oficial.

A continuació, veurem l'exemple de producció de missatges de tipus *broadcast* en què s'obre un *socket* per enviar els datagrames d'UDP a la xarxa LAN, tot utilitzant el port 11000. Aquest exemple envia un missatge de text específic que s'espera a l'altre costat.

```
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.");
    }
}
```

Finalment, veurem l'exemple complementari de codi on s'utilitza la classe *UdpClient* per rebre els datagrames UDP enviats a l'adreça de difusió al port 11000. En aquest cas, tots consumeixen els datagrames que arriben a aquest port en aquesta LAN (192.168.70.x) tot rebent el missatge de text enviat i produït per l'exemple 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. Processos

Un cop tenim implementada la lògica per establir connexions i poder enviar i rebre paquets, el següent pas és adaptar el codi perquè gestioni les connexions i l'espera dels diferents paquets que s'han de rebre. Per fer-ho, hi ha dues opcions diferents:

- **De manera asíncrona**. Simplement es revisa de tant en tant si ha arribat algun paquet. Si no n'ha arribat cap, seguim executant el nostre codi i després d'un temps determinat tornem a revisar si n'ha arribat algun. Això ens permet no haver d'esperar activament en el nostre codi l'arribada de nova informació.

- **De manera síncrona**. En aquest cas, quan mirem si ha arribat un paquet, bloquegem el sistema i esperem que arribi el primer paquet. El programa no se segueix executant si no és que disposem d'un sistema multifil.

Segons l'API que utilitzem per enviar i rebre missatges, tindrem o no disponible l'opció asíncrona. L'API bàsica que se'ns ofereix és la síncrona i, per això, per treballar eficientment amb *sockets* de manera síncrona, cal complementar el nostre joc amb un sistema multifil, tal com hem vist en els exemples anteriors. I això és així perquè en una comunicació sempre cal fer esperes per rebre informació de l'altre *host* (servidor o client), i aquest temps i recursos són crucials per a altres aspectes de l'execució, especialment quan es tracta de videojocs.

Sistema multifil

L'ús de diversos fils en un videojoc ens permet crear diferents tasques que estan treballant al mateix temps. En el cas d'ordinadors o consoles amb diverses CPU, la programació multifil ens permet treure el màxim profit de totes elles.

Un sistema multifil està molt relacionat amb el sistema operatiu. Per exemple, a Windows hem de treballar amb la seva pròpia API de multifil i en entorns UNIX, amb POSIX. En aquest punt ens interessa treballar amb eines que ens abstreguin del sistema operatiu per fer que el nostre codi sigui més portable, com ho fa Unity, per exemple.

Un altre punt interessant a tenir en compte és que, en diversos tipus de jocs, l'usuari exigeix fer ús d'altres **funcionalitats paral·leles**, com poder parlar per micròfon amb els altres jugadors. Per implementar aquest punt amb eficiència, hem de dedicar molts recursos i tenir amplis coneixements de xarxes i transferència de so (*streaming*) en temps real.

Una altra opció més recomanable és delegar aquest tipus de funcionalitats a un servei extern al nostre joc que ens permeti integrar-nos amb ell fàcilment sense necessitat d'incorporar explícitament el mecanisme ni les biblioteques requerides al nostre joc.

2.2. Formats de transmissió de dades

Com hem vist en els exemples de codi anteriors, per transmetre objectes entre instàncies de xarxa d'un joc multijugador, el joc ha de donar format a les dades dels objectes abans que puguin ser enviats per un protocol de la capa de transport. En aquest sentit, la serialització de dades d'objecte és només el primer pas en la transmissió d'estat entre els *hosts*. Una altra eina molt útil per minimitzar la mida de les dades transmeses i augmentar el rendiment és la compressió.

En aquest apartat també comentarem la tècnica de replicació que suporta la sincronització del món i de l'estat dels objectes entre els processos remots, i que sol realitzar-se mitjançant crides remotes a procediments que permeten abstreure la comunicació per crear codi d'alt nivell més comprensible i fàcil de mantenir.

2.2.1. Serialització

El terme serialització fa referència a l'acte de convertir un objecte contingut en memòria cap a una sèrie lineal de bits. Aquests bits es poden emmagatzemar al disc o enviar-se a través d'una xarxa, essent restaurats posteriorment al seu format original.

Per transmetre objectes entre instàncies de xarxa d'un joc multijugador, aquestes han de donar format a les dades que contenen els objectes a enviar de manera que puguin ser transmeses per un protocol de capa de transport.

El mecanisme de *stream* consisteix en un flux de dades; és una font de dades, normalment en bytes o caràcters, utilitzat tant l'escriptura com en la lectura de fitxers o també en la comunicació entre connexions de xarxa.

Un *stream* es fa servir bàsicament per abstreure'ns de les tasques de serialització explícita de les dades. Aquestes dades varien des d'estructures bàsiques com booleans o enters, fins a estructures de dades molt complexes que inclouen referències a altres estructures de dades.

En el nostre cas, utilitzem un *stream* de xarxa, que encapsula un *socket,* tot proporcionant accés als mètodes per enviar o rebre dades, en les seves diferents variants en funció del tipus de dades que es volen utilitzar.

Per il·lustrar aquest mecanisme, i com hem vist en l'exemple de TCP, podem obtenir del *socket* de la connexió un *stream* de xarxa, que ens permet escriure-hi a sobre i, en conseqüència, enviar per la xarxa les dades de la data i l'hora actuals.

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

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

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

Noteu, a més, que abans de l'enviament, aquesta informació ha estat codificada en format ASCII. La codificació és un mecanisme estretament relacionat amb la serialització de cadenes de caràcters o textos que permet codificar, transformar i interpretar de text a bits i a la inversa.

Encoding.ASCII

Aquesta classe permet obtenir una codificació del text proporcionat utilitzant el joc de caràcters ASCII.

2.2.2. Compressió

Com hem vist, amb les eines de serialització de dades és possible escriure codi que ens permeti enviar i rebre els objectes del joc a través de la xarxa. Tanmateix, no seria un codi eficient aquell que no respecta les limitacions d'amplada de banda, la quantitat de bits que es poden gestionar per unitat de temps imposades per la mateixa xarxa.

Les primeres connexions

En els primers temps dels jocs multijugador, els jocs van haver de conformar-se amb 2.400 bytes per segon o menys, depenent del tipus de connexió.

Avui, els desenvolupadors de jocs som més afortunats en gaudir de connexions d'alta velocitat en molts ordres de magnitud, però encara ens hem de preocupar de com utilitzar aquesta amplada de banda de la manera més eficient possible.

Posem, per exemple, un joc amb un món immens i centenars d'objectes en moviment. En aquest cas, l'enviament exhaustiu a temps real dels objectes potencialment accessibles per part d'un gran nombre de jugadors actius pot saturar la connexió sense importar la capacitat disponible de l'amplada de banda, ja que aquest ha de ser compartit per tots ells.

Per reduir aquesta problemàtica, s'ha de començar al nivell més baix, tot examinant les tècniques comunes per a la compressió de dades a nivell de bits i bytes. És a dir, quan un joc ha determinat que s'enviarà una peça específica de dades, permet realitzar aquesta operació utilitzant el mínim de bits possible.

Seguint l'exemple de codi del servidor TCP, abans d'enviar les dades a través de la xarxa, no només podem codificar-les, sinó que podem aplicar compressió per reduir la mida del paquet que enviarem:

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

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

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

L'exemple següent mostra un algoritme d'exemple que fa ús de la coneguda biblioteca *gzip* i un *stream* en memòria per 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 majoria del codi de serialització d'objectes segueix el mateix patró: es recorre cada variable membre de la classe d'un objecte i se serialitza el valor d'aquesta variable. Hi poden haver algunes optimitzacions, però en general el codi sol ser el mateix. De fet, és tan similar entre cada objecte que les eines de serialització permeten utilitzar un únic mètode d'escriptura i un altre de lectura per gestionar la majoria de les necessitats de serialització.

2.2.3. Replicació

La replicació d'un objecte implica quelcom més que enviar les seves dades serialitzades d'un *host* a un altre. Per tenir èxit, un joc de diversos jugadors simultanis ha de fer sentir als jugadors que estan interactuant en el mateix món. Quan un jugador obre una porta o acaba amb un enemic, tots els altres necessiten veure aquesta acció i els seus resultats.

Aquesta experiència compartida és un reflex de l'estat del món a cada *host* i implica intercanviar tota la informació necessària per mantenir la coherència entre l'estat de cada hoste. En primer lloc, un protocol de nivell aplicació ha de definir tots els possibles tipus de paquets, i el mòdul de xarxa ha d'etiquetar paquets que contenen dades d'objecte com a tal. Cada objecte té un identificador únic en el joc general (no només en la partida del jugador que l'ha generat), de manera que el *host* receptor coneix l'estat de l'objecte apropiat. Finalment, cada classe d'objecte necessita un identificador únic perquè el *host* receptor pugui crear un objecte de la classe correcta, si no n'existeix cap.

En els jocs a petita escala, és possible crear un món compartit entre *hosts* mitjançant la replicació de cada objecte en el món en cada paquet sortint. En canvi, en els jocs més grans, les dades de replicació no es poden ajustar per a tots els objectes en cada paquet, per la qual cosa s'ha d'utilitzar un protocol que admeti la transmissió dels canvis en l'estat del món. Cada canvi pot contenir accions de replicació per crear un objecte, actualitzar un objecte o destruir un objecte.

Per a més eficàcia, les accions d'actualització d'objectes poden enviar dades de serialització d'un subconjunt acotat de les variables de l'objecte. L'ús d'aquest enviament parcial depèn de la topologia de la xarxa global i la fiabilitat del protocol a nivell d'aplicació. En funció de la topologia de la xarxa del joc, hi ha diverses maneres de crear i fer complir la coherència entre els estats del món entre *hosts* remots.

Una manera seria mitjançant la **replicació total**. Aquest mètode, bastant comú, consisteix a tenir un servidor per transmetre l'estat del món a tots els clients connectats. Els clients reben aquest estat de transmissió i actualitzen el seu propi estat del món local. D'aquesta manera, tots els jugadors experimenten el mateix estat. L'estat del món es pot definir com l'estat de tots els objectes del joc en aquest món. Per tant, la tasca de transmetre l'estat del món es pot descompondre en la tasca de transmetre l'estat de cadascun dels seus objectes.

Replicació total

En la replicació total, quan el *host* receptor acaba de deserialitzar un objecte, aquest només utilitza una part de les dades.

En utilitzar codi de replicació per a múltiples objectes, ens podem plantejar realitzar la rèplica del món sencer, ja que és probable que sigui molt més fàcil d'aquesta manera. En aquest sentit, si la representació del món en un joc és prou petita, llavors tot l'estat sencer del món pot encaixar completament dins d'un únic paquet.

Una altra opció seria la **replicació parcial**, és a dir, enviar només els canvis de l'estat del món. Com que cada amfitrió manté la seva pròpia còpia, no cal replicar tot l'estat en un sol paquet. En comptes d'això, el remitent pot crear paquets que representen canvis en l'estat del món, i llavors el receptor pot aplicar aquests canvis al seu propi estat mundial. D'aquesta manera, un remitent pot utilitzar múltiples paquets per sincronitzar un món molt gran amb un *host* remot.

Replicació parcial

La replicació parcial només funciona si el remitent té una representació exacta de l'estat del món actual per determinar els canvis necessaris a replicar.

Quan s'envia una actualització d'objecte, és possible que el remitent no necessiti enviar cada propietat en l'objecte. El remitent pot serialitzar només el subconjunt de variables que han canviat des de l'última actualització. Per permetre-ho, es pot utilitzar un camp de bits per representar variables serialitzades. Cada bit pot representar una variable o un grup de propietats que se serialitzaran. Per eficiència i escalabilitat, aquesta última estratègia pot ser molt interessant.

Atès que Internet és inherentment poc fiable, no es pot suposar que l'estat del món del receptor es basi en els últims paquets transmesos per l'emissor. Per a aquest propòsit, els *hosts* haurien d'enviar paquets a través de TCP, fet que garantiria la fiabilitat; o alternativament, utilitzar un protocol de nivell d'aplicació dissenyat en la part superior d'UDP que proporcioni fiabilitat, com R-UDP.

Com podem intuir, la replicació d'objectes és un mecanisme clau en els jocs de diversos jugadors i serà un ingredient fonamental a l'hora de desenvolupar un videojoc de multijugador en xarxa depenent a més de les topologies de xarxa en què es basi.

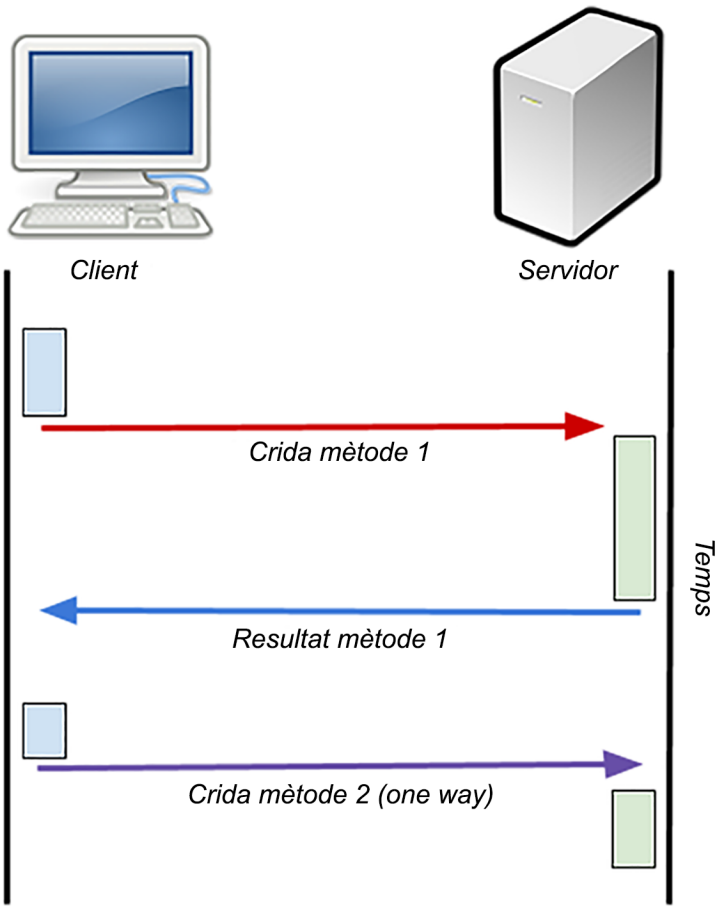
2.2.4. Crides remotes

Finalment, la **crida remota a un procediment** (RPC, *Remote Procedure Call*) és una eina que s'utilitza per abstreure'ns de com es realitza l'execució d'un procés allotjat en una altra màquina i s'espera la seva resposta amb el resultat d'aquesta execució.

El mecanisme d'RPC facilita una capa de programació per sobre de les llibreries de xarxa i evita els problemes més comuns (sincronització, enviament d'informació, etc.). És especialment útil en videojocs en què es requereix replicar més que l'objecte de dades d'estat entre les instàncies distribuïdes del joc.

Aquesta abstracció és àmpliament utilitzada per simplificar els protocols de comunicació, tant bidireccionals en crides a funcions en què s'espera una resposta, com en comunicacions unidireccionals (*one way*) en què no s'espera cap resposta de la invocació remota.

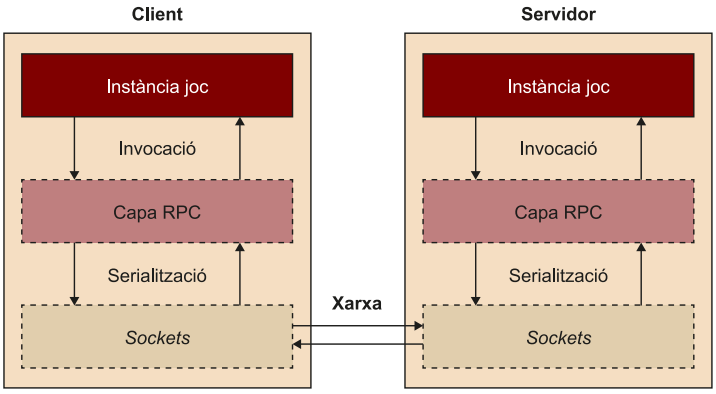
Figura 2. Comunicació bidireccional i unidireccional (*one way*)



Un exemple d'aquest últim cas, en el qual es vol transmetre informació no relacionada amb l'estat de l'objecte i sense retorn, podria ser l'ordre d'una acció remota en un altre *host* (per exemple, produir el so d'una explosió). Com que en el *host* remot ja ha d'existir un objecte responsable de realitzar aquesta acció amb el seu *script* associat, allò que interessa és clarament executar el codi remot de la manera més simple possible. Ho veurem més clar quan estudiem l'exemple de *Tanks!* que treballarem més endavant en els reptes d'aquest mòdul.

Així doncs, una crida a un procediment remot s'executa en un *host* origen, que causa una reacció en un procediment d'un altre *host,* essent cada invocació una crida única, tot indicant l'identificador de l'objecte remot amb les seves variables associades enviades com a paràmetres, i essent serialitzades en origen i deserialitzades en destí.

Figura 3. Crida a un procediment remot



Recomanació

Com es recomana altres vegades, si hi ha l'opció de fer-ho, useu la capa d'RPC que ens proporcioni el *framework* utilitzat, com és el cas d'Unity.

Noteu que aquesta abstracció també ens permet no haver de guardar la informació explícita dels *hosts* dels altres jugadors o del servidor, sinó que es basa en centrar-nos en la informació dels objectes remots que, s'executin on s'executin, seran igualment invocats gràcies al sistema subjacent.

Hi ha tant la possibilitat d'utilitzar un dels diversos protocols coneguts de nivell d'aplicació disponibles (per exemple, XML-RPC), com l'opció de construir la nostra pròpia capa d'RPC.

3. Tipologies de joc de xarxa

Hi ha diferents maneres de connectar els diversos *hosts* o instàncies de jugador del nostre joc. La forma més clàssica és separar les connexions segons el rol que té cadascun dels *hosts* que intervenen en la comunicació. En el cas que vulguem una jerarquia de comunicació, necessitarem un sistema client-servidor, i en el cas que vulguem que totes les instàncies es comportin de la mateixa manera, necessitarem una arquitectura *peer-to-peer*.

3.1. Client-servidor

En aquest tipus de comunicació, definirem una instància com a servidor i la resta com a clients:

- El **servidor** és el que roman a l'espera de peticions de clients i les va servint a mesura que arriben.
- El **client** és responsable de realitzar peticions al servidor.

Per exemple, cada vegada que un client vol enviar un missatge o alguna informació als altres clients, aquesta informació l'envia a través del servidor, essent el servidor el que la distribueix entre els clients destí.

Jocs client-servidor

Els exemples són *World of Warcraft* o *Clash of Clans*.

Aquests són alguns avantatges que obtenim quan s'utilitza un sistema client-servidor:

- La seguretat es pot controlar millor, ja que el servidor pot detectar si algun dels clients està realitzant alguna acció inapropiada.
- En el cas que hi hagi dades persistents, permet garantir la seva integritat.
- Normalment, els servidors són sistemes dedicats amb un hardware específic, cosa que augmenta el seu rendiment en comparació amb ordinadors estàndard.
- Com que és un sistema centralitzat, és molt més fàcil gestionar els usuaris i les partides.

D'altra banda, aquests són alguns dels inconvenients associats a l'ús d'un sistema client-servidor:

- Hem de crear dos tipus diferents de lògica, una que s'encarregui de les tasques pròpies del servidor, com ara la gestió del joc; i una altra del client, que ens permeti jugar-lo.

- Si volem utilitzar un servidor dedicat hem de considerar l'alt cost del hardware. A més, es necessiten administradors de xarxa i tècnics de sistemes que s'encarreguin de configurar i mantenir actualitzat el servidor, i garantir el seu bon funcionament.

- En el cas que hi hagi un problema al servidor, s'interromp completament el desenvolupament del joc.

- La connexió del servidor pot suposar un coll d'ampolla en el rendiment global del sistema.

Finalment, per estructurar un joc en xarxa en el qual els jugadors estan connectats directament al servidor sense conèixer la resta de clients, ens trobem amb les dues aproximacions contraposades.

La primera aproximació es coneix com a ***authoritative server***. Aquesta aproximació requereix d'un servidor per realitzar la simulació de tot el món, l'aplicació de les regles del joc i el processament de les interaccions provinents de tots els clients. Cada client envia les seves interaccions al servidor i rep contínuament actualitzacions d'estat, que mai modifica per si mateix. Això permet al servidor escoltar tots els clients i posteriorment decidir com s'actualitza l'estat.

Per això pot observar una diferència clara entre el que el jugador vol realitzar i el que realment succeeix. Un clar avantatge respecte a evitar trampes dels clients, ja que, per exemple, els clients no poden enganyar el servidor sobre l'estat del joc (com la falsa mort d'un altre jugador), ja que és el servidor qui ho ha de decidir.

Un desavantatge potencial d'aquesta estratègia és el temps que triguen els missatges a viatjar entre tants nodes, la possible congestió o el coll d'ampolla que això provoca, o confiar en un punt únic de fallida (SPOF, *Single Point Of Failure*).

SPOF

És un component que, després d'un error en el seu funcionament, provoca una fallida global en tot el sistema.

En contraposició amb aquesta estratègia, ens trobem la coneguda justament com a ***non-authoritative server***. Aquesta vegada, el servidor no s'encarrega de controlar el resultat de cada interacció del jugador. Són els mateixos clients que processen localment les entrades i la lògica del joc, i després envien el resultat de qualsevol acció determinada al servidor. Tot seguit, el servidor sincronitza totes les accions de l'estat del món que rep.

Aquesta segona opció fa més simple la implementació del servidor, que queda relegat a tasques de retransmissió de missatges sense lògica de processament associada, essent els clients els propietaris reals dels seus objectes i els únics autoritzats a enviar actualitzacions d'aquests objectes a través de la xarxa.

Actualment, molts dels jocs que utilitzen l'arquitectura client/servidor es basen en una aproximació híbrida d'ambdues estratègies, on alguns comportaments es tracten d'una manera autoritativa i d'altres no. Cal tenir en compte que si disposem de certes característiques del joc que són sensibles a les trampes (*cheating*), la lògica associada a prevenir-les haurà de ser tractada de forma autoritativa (centralitzada en el servidor) mentre que altres característiques menys importants es poden tractar de forma no autoritativa (en els clients), tot reduint així el coll d'ampolla del servidor i incrementant el rendiment del joc.

3.2. *Peer-to-peer*

En un sistema *peer-to-peer* (o d'igual a igual), totes les instàncies es troben en el mateix nivell jeràrquic, és a dir, es considera que tots els *hosts* són iguals. Per tant, en un joc basat en un sistema *peer-to-peer* no hi ha cap instància que tingui més control del joc que les altres, ni hi ha cap mediador que s'encarregui de distribuir l'estat del joc o d'enviar missatges entre els diferents clients.

Jocs *peer-to-peer*

Alguns exemples són *Age of Empires* o *Advance Wars*.

Alguns dels avantatges que proporciona un entorn *peer-to-peer* són els següents:

- No cal invertir en recursos com servidors específics o tècnics de suport, ja que cada jugador actua com a administrador del seu propi equip.

- Implementa una sola lògica que serveix a la vegada de client i servidor i és comuna per tots, la qual cosa pot facilitar el desenvolupament del joc i el disseny de la seva arquitectura interna.

- No es depèn d'un sistema central, fet que evita problemes de connexió i colls d'ampolla. Per tant, si un client no funciona, això no afecta la resta dels jugadors.

D'altra banda, els sistemes *peer-to-peer* tenen els següents inconvenients:

- Són poc escalables. Es poden utilitzar sense problemes a petita escala, però si provem d'utilitzar-los amb molts clients distribuïts el rendiment generalment es veu afectat negativament.

- La manca d'un sistema central fa que les possibilitats de desincronització entre clients siguin molt més grans, tot provocant possibles incongruències en allò que dos clients poden estar veient al mateix temps.

- La seguretat d'aquest tipus de sistemes és molt baixa, ja que hem de confiar que els altres clients siguin fiables.

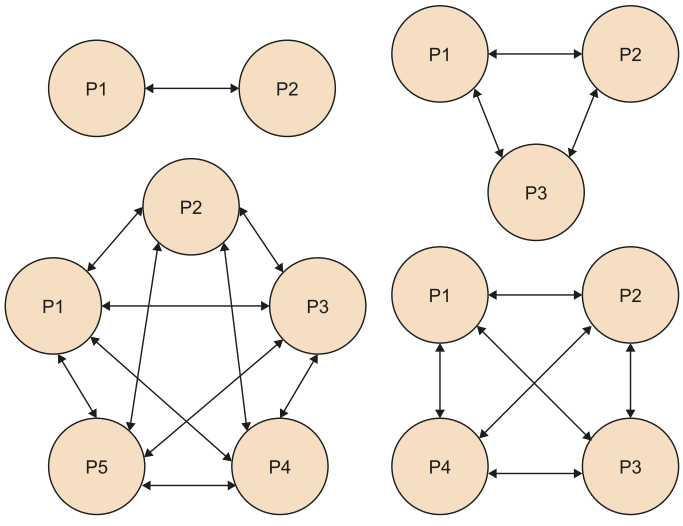
Les primeres implementacions de sistemes *peer-to-peer* utilitzaven la tècnica de difusió o *broadcast* (enviament simultani a tots els nodes) de les xarxes LAN. A mesura que la tecnologia de xarxa va anar evolucionant, aquests sistemes han deixat d'utilitzar-se, bàsicament perquè a Internet és inviable fer *broadcasting* de la mateixa manera que es fa a les xarxes LAN.

Taula 2. Correspondències entre jugadors, tipus de xarxa, connexions, enviaments i recepcions

Nombre de jugadors / Tipus de xarxa	***Broadcast***	***Peer-to-Peer***	**Client / Servidor**
Nombre de connexions	0	$\sum_{x=1}^{N-1} x$	Client = 1 Servidor = N
Enviaments	1	N-1	Client = 1 Servidor = N
Recepcions	N-1	N-1	Client = 1 Servidor = N

Un sistema *peer-to-peer* s'implementa de manera que es creen diferents connexions equivalents al cas client/servidor entre les diferents instàncies del joc. En el cas que el nombre de jugadors creixi, s'haurien de fer massa connexions, cosa que pot convertir el sistema en inviable.

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



Això vol dir que en el cas de dos jugadors necessitarem una única connexió, essent òptim respecte a l'esquema client-servidor. Sense entrar a comparar aquest escenari, si seguim augmentant el nombre de jugadors, aquesta opció empitjora ràpidament. En el cas de tres jugadors seran dues connexions per jugador, tres en total. Amb quatre jugadors passem a necessitar tres connexions per jugador, esdevenint-ne sis en total. I ja, amb cinc jugadors, el nombre de connexions s'eleva a quatre per jugador, és a dir, vuit connexions en total.

Molts jugadors

Si voleu implementar un joc amb un nombre de jugadors elevat, cal utilitzar tècniques o estructures que estalviin i comparteixin totes les connexions possibles per minimitzar aquest problema tant com sigui possible.

3.3. Classificació

Recopilant els conceptes que hem vist fins ara i segons la interacció entre jugadors, els videojocs es poden classificar de la següent manera:

- **Jocs per torns**. Els jugadors intervenen en el joc de manera alternativa, no simultània. En aquest tipus de jocs, la percepció del món del joc que tenen els diferents jugadors pot ser exactament la mateixa.

- **Jocs interactius**. Els jugadors interactuen simultàniament amb el mateix món del joc. En aquest tipus de jocs, mantenir la consistència absoluta entre els jugadors sol ser impossible a la pràctica.

Segons la interacció del client amb el servidor, els jocs es poden classificar de la manera següent:

- **Jocs volàtils**. Es juguen en partides que comencen i acaben en un lapse de temps determinat. Si el servidor cau o es talla la connexió, el jugador entén que el progrés es perd. Tanmateix, molts jocs volàtils tenen una petita part de persistència remota per a *rankings,* puntuacions, etc.

- **Jocs al núvol**. Un servidor (o diversos) actua com un repositori de dades, gestor de missatgeria i proveïdor d'altres serveis, tot dotant de persistència les accions dels jugadors. El progrés del jugador s'ha de conservar. Normalment s'utilitza aquesta terminologia per a jocs basats per torns, basats en dades i comandaments simples.

- **Jocs massius**. Una xarxa de servidors que mantenen un món persistent i es comuniquen amb els clients. El progrés del jugador s'ha de conservar.

Finalment, aquestes classificacions s'exemplifiquen amb un títol molt conegut per a cadascuna d'elles.

Taula 3. Classificacions de jocs amb exemples

	Peer-to-peer	**C/S volàtils**	**C/S al núvol**	**C/S massius**
Per torns	*Advance Wars*		*Clash of Clans*	
Interactius	*Age of Empires*	*Battlefield*		*World of Warcraft*

4. Tècniques de millora del joc en xarxa

Videojocs en línia

Anomenem videojocs en línia als videojocs que es poden jugar a través d'Internet, independentment de la plataforma.

El moment que es realitza la fase de proves d'un videojoc en xarxa és quan es manifesten una sèrie de factors negatius que no estan presents en un entorn minuciosament controlat, com ara una xarxa local. Atès que la feina del desenvolupador és anticipar-se a tots aquests problemes i fer que no siguin visibles per a l'usuari, abans de llançar un videojoc en línia, i especialment durant la seva fase *beta*, s'han de solucionar els problemes coneguts i aplicar millores de rendiment, així com les mesures de seguretat.

4.1. Problemes en la transmissió de dades

Com ja hem vist, els videojocs en línia es troben en un ambient típicament hostil, ja que han de competir per l'amplada de banda per enviar paquets als servidors i els clients es troben repartits per tot el món. Això produeix, a diferència dels entorns LAN, que en els jocs en línia sigui freqüent haver de fer front a la pèrdua de dades i els retards. En aquest apartat s'exploren alguns dels problemes coneguts quan es desenvolupen jocs en xarxa per buscar possibles solucions.

4.1.1. Latència

El primer d'aquests factors és la latència. La paraula *latència* té diferents significats en diferents situacions.

> En el context dels jocs d'ordinador, es refereix a la quantitat de temps entre una causa i el seu efecte observable.

Depenent del tipus de joc, això pot succeir en situacions molt diverses: des del període de resposta entre un clic del ratolí i l'acció d'una unitat en un joc d'estratègia, fins al període comprès entre el moment que un usuari realitza una acció amb un dispositiu d'entrada –com prémer un botó– i que el vehicle del jugador en un joc de cotxes realment es mogui.

Una certa quantitat de latència és inevitable i diferents gèneres de jocs tenen diferents llindars de latència acceptables. Per exemple, els jocs de realitat virtual solen ser els més sensibles a la latència, a causa dels seus estrictes requeriments de temps real. En aquests casos, es requereix una latència de menys de 20 mil·lisegons per no produir efectes negatius sobre l'experiència i la salut del jugador. En els jocs de lluita, d'acció en primera persona o altres jocs d'acció

ràpida també hi ha una gran sensibilitat a la latència, que pot variar de 16 a 150 mil·lisegons abans que l'usuari comenci a notar, independentment de la velocitat de fotogrames, que el joc és lent o que no respon correctament.

Per tant, la disminució de la latència és una bona manera de millorar l'experiència de joc i, per a això, es requereix entendre els factors que l'afecten.

Entre altres causes, el retard experimentat per un paquet mentre viatja des d'un *host* d'origen a la seva destinació és sovint la font principal de latència en jocs multijugador. Podem separar en quatre classes principals el retard que un paquet experimenta durant la seva vida útil:

NAT (*Network Address Translation*)

És un mecanisme utilitzat per encaminadors IP per intercanviar paquets, basat en convertir en temps real les adreces utilitzades en els paquets transportats.

- **Retard de processat**. Un encaminador s'encarrega de llegir els paquets de la interfície de xarxa, examinar l'adreça IP de destí, resoldre la màquina de destí que ha de rebre el paquet i, finalment, transmetre'ls a través de la interfície adequada. Aquest processament implica un retard que pot ser incrementat en incloure qualsevol funcionalitat addicional que l'encaminador proporcioni, com ara NAT (*Network Address Translation*) o xifrat.

- **Retard de transmissió**. Perquè un encaminador transmeti un paquet, ha de tenir una interfície de capa d'enllaç que permeti que el paquet es transmeti a través d'algun mitjà físic. El protocol de capa d'enllaç controla la velocitat mitjana a la qual els bits es poden escriure en el mitjà, cosa que implica un temps necessari per dur a terme aquesta operació.

- **Retard d'espera**. Un encaminador només pot processar un nombre limitat de paquets a la vegada. Si els paquets arriben a un ritme més ràpid que el de processament, entren en una cua de recepció, tot esperant a ser processats. De la mateixa manera, una interfície de xarxa només té sortida per a un sol paquet a la vegada. Així doncs, després que un paquet sigui processat, si la interfície de xarxa corresponent està ocupada, entra en una cua de transmissió.

- **Retard de propagació**. Independentment del mitjà físic, la majoria de les vegades la informació no pot viatjar més ràpid que la velocitat de la llum. Amb fibra òptica s'aconsegueix una velocitat superior a la meitat de la velocitat de la llum, però no la seva velocitat completa. Les connexions a través d'«aire», com ara de les xarxes mòbils o las de satèl·lit, tenen una latència molt elevada.

Alguns d'aquests retards poden ser pal·liats amb certes millores. A més, el retard en el processament normalment és un factor menor. Avui, la majoria dels processadors d'encaminadors són molt ràpids.

Ping

És una utilitat diagnòstica en xarxes de computadors que comprova l'estat de la comunicació entre dos punts remots i s'utilitza per mesurar la latència entre ells.

Aquestes millores poden ser mesures de baix nivell, com la reducció de capçaleres en la mida dels paquets per reduir el retard de transmissió, o reduir al mínim el processament per ajudar a minimitzar les cues de retard, o mesures d'alt nivell, més relacionades amb les decisions que podem prendre com a desenvolupadors de videojocs.

El retard de propagació és sovint un bon objectiu per a la millora de la latència, ja que es basa en la distància entre *hosts*. La millor manera d'optimitzar seria situar més a prop els amfitrions (*locality*) segons la topologia.

Matchmaking

Anomenem *matchmaking* al procés d'agrupar i enfrontar jugadors en sessions de joc en línia.

- En els jocs *peer-to-peer*, això significa prioritzar la localitat geogràfica en el temps de *matchmaking*.

- En els jocs client-servidor, això significa desplegar servidors de jocs segur que estiguin disponibles a prop dels diferents jugadors.

A vegades, la localització física no és suficient per garantir el retard de baixa propagació: pot ser que no hi hagi connexions directes entre localitzacions, fet que requereix encaminadors per conduir el trànsit per rutes tortuoses i no a través d'una línia directa. És important tenir en compte les rutes existents i les futures en la planificació de les ubicacions dels servidors de joc.

Tanmateix, en alguns casos no és factible dispersar els servidors de joc a través d'una àrea geogràfica, ja que es vol que tots els jugadors de tot un continent puguin jugar entre ells. Casos com *League of Legends* es van trobar en aquesta situació: la dispersió de servidors de joc per tot el país no era una opció i van decidir executar l'estratègia inversa. Per fer-ho, van construir la seva pròpia infraestructura de xarxa, tot analitzant els proveïdors de serveis d'Internet (ISP) dels Estats Units, per assegurar-se que podien controlar les rutes de circulació i reduir la latència de la xarxa tant com fos possible.

Com que la *latència* és un terme molt utilitzat, els desenvolupadors de jocs solen discutir més sovint el concepte específic del temps d'anada i tornada (*Round Trip Time*, RTT) per condensar en una única mètrica els quatre tipus de retard indicats.

RTT (*Round Trip Time*)

Es refereix al temps que triga un paquet a viatjar d'un *host* a un altre, i després el paquet de resposta a recórrer tot el camí de tornada.

Això acaba reflectint no només els retards de processament, la gestió de cues, transmissió i propagació de dues rutes, sinó també els FPS (*frames per second*) del receptor, és a dir, que redueix la velocitat de fotogrames de la màquina remota, ja que també afecta el procés d'enviament de paquets de resposta. A

més, el trànsit no té necessàriament la mateixa velocitat de viatge en cada direcció. L'RTT rarament es duplica a partir del temps que triga un paquet a passar d'un hoste a un altre, ja que el protocol TCP/IP no garanteix que el camí sigui el mateix per a ambdós trajectes. Tot i això, és una pràctica habitual utilitzar com a aproximació el càlcul de la meitat de l'RTT com a temps de viatge d'una sola ruta.

4.1.2. *Jitter*

Amb una bona estimació de l'RTT ja es poden adoptar mesures per mitigar aquests retards i oferir als jugadors la millor experiència possible. Tanmateix, quan s'escriu codi de xarxa, cal tenir en compte que l'RTT no és ni molt menys un valor constant. Per a qualsevol dels dos *hosts*, l'RTT entre ells no s'estableix normalment al voltant d'un valor determinat basat en els retards mitjans involucrats.

Es coneix com a *jitter* la variabilitat del temps d'arribada de dos paquets consecutius. Aquest efecte és especialment molest en els jocs multijugador en línia, ja que provoca que alguns paquets arribin massa aviat o massa tard, fet que impedeix lliurar-los a temps.

Qualsevol dels retards involucrats pot canviar amb el temps, fet que pot provocar una desviació en RTT del valor esperat i contribuir a la fluctuació, encara que alguns són més propensos a produir variacions que altres:

- **Retard de processament**. En ser el component menys significatiu de la latència de xarxa és també el menys significatiu, tot contribuint al seu desfasament. Pot variar en com els encaminadors ajusten les rutes dinàmicament en viatjar els paquets, però és una preocupació menor.

- **Retards de transmissió i propagació**. Estan molt relacionats, ja que els protocols de capa d'enllaç determinen el retard de la transmissió i la longitud de la ruta determina el retard de propagació. Així doncs, canvien quan els encaminadors equilibren de manera dinàmica la càrrega de trànsit i alteren les rutes per evitar les zones congestionades. Això pot fluctuar ràpidament durant els moments de trànsit pesat, de manera que els canvis de ruta poden alterar significativament els temps d'anada i tornada.

- **Retard d'espera**. Es relaciona amb el nombre de paquets que un encaminador pot processar. A mesura que el nombre de paquets que arriben a un encaminador varia, el retard de cua també varia. Les ràfegues de trànsit pesat poden provocar retards d'espera significatius i variar els RTT.

A més, el *jitter* pot afectar negativament els algoritmes de reducció de RTT, i el que és pitjor, pot fer que els paquets arribin completament desordenats.

4.1.3. Pèrdua de paquets

Més important que la latència, l'RTT i el *jitter* és la pèrdua de paquets, el problema més gran que afronten els desenvolupadors de jocs de xarxa. Clarament, el problema ja no és que un paquet trigui molt de temps a arribar al seu destí, sinó que el paquet no arribi mai.

Els paquets es poden perdre per diversos motius.

- **A causa d'un mitjà físic poc fiable**. En el seu origen, la transferència de dades és la transferència d'energia electromagnètica. Qualsevol interferència electromagnètica externa pot provocar la corrupció d'aquestes dades. En el cas de les dades corruptes, la capa d'enllaç detecta la corrupció en validar les sumes de comprovació (per exemple, CRC) i descarta les trames que contenen errors.

CRC (*Cyclic Redundancy Check*)

És un codi de detecció d'errors el càlcul del qual és una divisió en què la resta ha de coincidir.

- Les capes d'enllaç tenen regles sobre quan poden i no poden enviar dades. A vegades, un canal de capa d'enllaç és completament ple, i s'ha de descartar una trama de sortida. Com que la capa d'enllaç no proporciona cap garantia de fiabilitat, aquesta és una resposta perfectament acceptable.

- **Una capa de xarxa poc fiable**. Recordem que quan els paquets arriben a un encaminador, es col·loquen en una cua de recepció. Aquesta cua té un nombre màxim de paquets que pot contenir. Quan la cua és plena, l'encaminador comença a descartar tant paquets que ja són a la cua com paquets entrants. Per tant, la pèrdua de paquets perduts és un fet que no es pot evitar, i la seva arquitectura de xarxa ha de ser dissenyada tenint en compte aquest problema.

Independentment de la mitigació de la pèrdua de paquets, l'experiència de joc serà millor amb un nombre menor de paquets perduts. Un cop més, la solució d'utilitzar un centre de dades amb servidors tan a prop dels seus jugadors com sigui possible és la solució més directa i fiable. Això és perquè implica menys encaminadors i cables, i per tant, una probabilitat menor que un d'ells descarti les nostres dades.

Una altra aproximació consisteix a enviar el menor nombre de paquets com sigui possible. Molts encaminadors calculen la seva capacitat de la cua en funció del recompte de paquets, no de la quantitat total de dades. En aquests casos, un joc té més probabilitat d'inundar encaminadors i desbordar les cues si envia molts paquets petits que si ho fa amb un número menor de paquets grans. L'enviament de deu paquets de 100 bytes a través d'un encaminador

obstruït requereix deu ranures lliures a la cua per evitar la pèrdua de paquets. Tanmateix, l'enviament dels mateixos 1.000 bytes en un únic paquet només requereix una ranura de cua lliure.

No obstant això, no tots els encaminadors basen l'adjudicació de les seves ranures en el nombre de paquets; alguns es basen en l'amplada de banda entrant i llavors resulta més beneficiós enviar paquets petits.

Quan les seves cues són plenes, un encaminador no necessàriament deixa caure cada paquet entrant, en comptes d'això pot descartar un paquet que espera a la cua. Això succeeix quan l'encaminador determina que el paquet entrant té més prioritat o és més important que el de la cua. Els encaminadors prenen decisions de prioritat en funció de dades de qualitat de servei (*Quality of Service*, QoS) que es troben a la capçalera de la capa de xarxa, o també a partir de la informació obtinguda mitjançant la inspecció de la càrrega útil de paquets.

Alguns encaminadors estan configurats fins i tot per prendre decisions ràpides i destinades a reduir el trànsit en general. Per exemple, a vegades es deixen caure els paquets UDP abans que els paquets TCP perquè aquests acaben essent enviats de manera automàtica.

La comprensió de les configuracions de l'encaminador al voltant dels centres de dades, i dels ISP en tot el mercat de destí, pot ajudar a ajustar els tipus de paquets i els patrons de trànsit per reduir la pèrdua de paquets. En última instància, la manera més fàcil de reduir la pèrdua de paquets és assegurar-se que els servidors utilitzin connexions a Internet ràpides, estables i tan a prop dels seus clients com sigui possible.

4.2. Millora del rendiment

Una vegada estudiats els problemes, procedirem a descriure les solucions per millorar el rendiment en els videojocs en línia.

4.2.1. La infraestructura

La primera opció que tenim per millorar el nostre sistema és el hardware, el sistema operatiu i la connexió a Internet. És molt important que el nostre sistema estigui preparat per poder enviar i rebre totes les dades que necessitem. Per tant, cal utilitzar una connexió que ens permeti millorar alguns d'aquests aspectes:

- Reduir al mínim el nombre de paquets perduts.

- Reduir al mínim el temps de latència, és a dir, el temps que triga un paquet a viatjar entre l'origen i el destí.

- Reduir al màxim el *jitter*, és a dir, la diferència entre el temps d'arribada de dos paquets consecutius. Ens interessa mantenir un ritme constant per poder integrar coherentment el flux d'entrada/sortida en el joc.

Aquestes millores normalment es realitzen invertint més diners en infraestructura, ja que no es poden realitzar mitjançant programació.

4.2.2. Protocol

Un altre element clau per millorar la fluïdesa d'un joc és l'optimització del protocol de comunicacions utilitzat. Oferirem alguns consells sobre com es pot millorar la mida dels paquets i la quantitat de paquets que s'envien:

- Enviar només els canvis d'estat en comptes d'enviar contínuament l'estat actual. Per exemple, en comptes d'enviar la nostra posició contínuament, només enviem l'ordre d'avançar, retrocedir, etc. Aquest procés requereix que cada cert temps s'enviï algun paquet de sincronització per garantir que ambdós sistemes es troben en el mateix estat.

- Triar una àrea al voltant del personatge i enviar únicament els esdeveniments que es produeixin dins d'aquesta zona. També podem establir diferents seccions més grans en què podem enviar menys informació (per exemple, només posicions de personatges per dibuixar el mapa).

- Intentar condensar el màxim d'informació en un únic paquet de dades. Més paquets impliquen un major *overhead* (ja que enviem més capçaleres), i per tant generem més trànsit de xarxa del que és necessari.

- Missatgeria jeràrquica. Es tracta d'adaptar la quantitat d'informació enviada segons les diferents velocitats dels clients (podem tenir des de línies de fibra o ADSL de tot tipus de velocitats i simetries). La tècnica consisteix a assignar una certa prioritat als missatges que enviarem i decidir si ho transmetrem o no en funció de la importància que tingui per al funcionament correcte del joc. Per exemple, els canvis de posició tenen una prioritat molt alta, però un canvi en l'animació de l'usuari és de prioritat molt baixa.

4.2.3. A títol predictiu

Un altre dels mètodes més freqüentment utilitzats perquè el jugador no noti els problemes que hi pugui haver a la xarxa és l'ús de la predicció del comportament dels elements del joc.

D'aquesta manera, el client prova de predir quin serà el següent moviment que faran els objectes que es troben a la seva zona. Cada vegada que rep una posició i una orientació del servidor, simplement hem de corregir parcialment la posició dels objectes que havíem intuït perquè s'ajustin a la realitat. Es recomana utilitzar algoritmes que facin aquesta transició d'una manera simple, perquè no veiem canvis bruscos en les posicions dels elements.

Figura 5. Algoritme de predicció de *Counter-Strike*

Un algoritme de predicció necessita dos components principals:

- Les dades històriques de què han fet tots els elements en els últims informes del servidor, i
- un bon sistema de predicció que permeti anticipar el moviment real dels objectes.

Tanmateix, no hi ha cap sistema que sigui capaç de predir sempre les accions dels altres usuaris i dels elements del sistema, per la qual cosa hem de tenir clar que el resultat de la nostra predicció no és mai fiable al cent per cent.

En alguns jocs en línia, la predicció pot produir situacions bastant estranyes o còmiques. Tot i que la sensació d'estar jugant és bastant contínua, si hi ha hagut un tall a la xarxa pot ser que apareguem de sobte en un punt on érem feia una bona estona. Això és perquè, quan es recupera la connexió, veiem que hi ha una gran diferència entre l'estat del client i el del servidor, de manera que el servidor ens situa de nou en l'última posició vàlida que coneixia.

En alguns jocs aquest problema pot ser més preocupant, ja que si la connexió es talla en un moment clau (en un combat, per exemple), quan el servidor ens actualitza l'estat podem descobrir que el nostre personatge ha mort. En aquests casos ens serveix de ben poc la predicció de les accions.

4.2.4. En l'enviament d'accions

En alguns tipus de jocs, sobretot els que requereixen molt moviment d'elements (com ara els RTS), no és factible enviar les posicions i l'estat de tots els elements a cada moment, ja que això suposaria una despesa considerable d'amplada de banda. En comptes d'això, s'envien les ordres que donen els jugadors i a quins elements del joc els les donen.

Perquè aquest sistema funcioni, tots els clients han d'executar la mateixa versió del joc, a la mateixa velocitat d'execució (normalment es treballa a 10 fps) i aquesta ha de ser totalment determinista. És a dir, si un jugador dona l'ordre a un comandament de moure 100 unitats d'A a B, tots els clients han de calcular la mateixa ruta per a les 100 i començar a moure-les al mateix temps.

Garantir el determinisme

Un dels problemes més importants que s'ha d'abordar per garantir el determinisme és la resolució dels números de coma flotant.

L'element clau en el sistema és la sincronització inherent. Tot i que no s'envien missatges, tots els sistemes s'han de sincronitzar en tots els cicles d'execució. Cada cert temps s'ha de realitzar un test de sincronització entre tots els elements del joc (per exemple, comprovar que totes les unitats es troben en la mateixa posició i/o en el mateix estat). Un únic error de sincronització no té solució i comportarà la finalització de la partida.

4.3. Seguretat

La seguretat en la comunicació és necessària per garantir el correcte desenvolupament d'un joc. La seguretat és important per tres motius principals:

- Per protegir la informació sensible dels usuaris (contrasenyes, etc.).
- Per evitar suplantacions de personalitat (un usuari es fa passar per un altre).
- Per facilitar un entorn de joc just, en el qual cap usuari no pot pugui avantatjar els altres amb l'ús de trucs o trampes.

La protecció de la connexió es pot realitzar mitjançant diverses tècniques que actuen a diferents nivells de l'arquitectura TCP/IP:

- Pel que fa a la xarxa o el transport, podem filtrar paquets sospitosos o maliciosos. En aquest àmbit podem detectar i prevenir alguns dels atacs de *hackers* que proven de modificar adreces i ports per enganyar els servidors.
- Pel que fa a l'aplicació, podem protegir les dades tot xifrant la informació continguda en els paquets. Només és aconsellable aplicar aquesta tècnica en moments concrets i quan es tracti d'informació sensible, ja que el procés de xifrat i desxifrat requereix certa computació que afectaria el desenvolupament d'una partida.

Hi ha serveis en línia que proporcionen mecanismes per millorar la seguretat dels jocs i impedir que els usuaris utilitzin trampes per obtenir avantatge respecte als altres usuaris.

En aquest sentit, si optem per utilitzar una plataforma de serveis, alguns dels que s'ofereixen són protegir els videojocs d'algunes de les trampes més conegudes, com per exemple falsejar les estadístiques o els marcadors. Aquestes plataformes ofereixen un entorn tancat en el qual hi ha un control més exhaustiu dels jocs que se suporten.

A més, aquest tipus de serveis mantenen una base de dades sobre els usuaris «banejats» (*banned*), corresponent als jugadors que han estat descoberts provant de realitzar alguna acció il·legal i, en conseqüència, ja no se'ls permet accedir al joc, com a càstig.

5. Desenvolupament en Unity

El motor de joc Unity va ser llançat per primera vegada el 2005. Els darrers anys s'ha convertit en un motor de joc molt popular utilitzat per molts desenvolupadors. Aquest motor ofereix algunes funcions de sincronització i RPC molt útils. Concretament, a partir d'Unity 5.1 es va introduir una nova biblioteca de xarxa anomenada UNet.

5.1. UNet

La biblioteca UNet proporciona dues API diferents:

- HLAPI (*high-level API*), de nivell superior, capaç de gestionar la majoria dels casos d'ús d'un videojoc en xarxa pel que fa als objectes de joc i les seves interaccions.

- LLAPI (*low-level API*), de transport de baix nivell, per sota de l'anterior, està pensada per ser utilitzada com a capa de comunicació d'una manera molt semblant al mecanisme de *sockets*.

Descriurem amb més detall l'API de nivell superior, essent LLAPI relegada a una manera avançada d'afinar els mecanismes de xarxa en cas que això fos necessari.

5.1.1. UNet HLAPI

En Unity, cada classe *GameObject* s'utilitza com a recipient per a diversos components, amb tots els comportaments delegats. Això permet una millor delimitació entre diferents aspectes del comportament d'un objecte de joc, tot i que si hi ha dependències entre múltiples components fa que el desenvolupament d'aquests sistemes sigui encara més complex.

En aquesta API d'alt nivell, Unity proporciona un administrador de xarxa (la classe de ***NetworkManager***) per encapsular l'estat d'un joc en xarxa. Aquest administrador de xarxa pot funcionar en tres modes diferents: com a client independent, com a servidor independent (dedicat), o com a ***host***, tot combinant la idea de client i servidor al mateix temps.

Atès que Unity utilitza la topologia client-servidor, el mecanisme de *spawn* és molt diferent del mecanisme anàleg en un videojoc d'un sol jugador en local. Concretament, quan es genera un objecte en el servidor a través de la funció ***NetworkServer.Spawn***, significa que aquest objecte de joc serà rastrejat pel servidor mitjançant un identificador de xarxa.

Spawn

És un mecanisme d'instància dinàmica que permet crear *GameObjects* en temps d'execució a partir d'un *prefab.*

A més, aquest objecte ha de ser replicat i generat en tots els clients al mateix temps. Per poder realitzar aquesta operació, és un requisit indispensable registrar el *prefab* (configuració preestablerta) del *GameObject* implicat. Un cop l'objecte és generat en el servidor, les propietats es poden replicar en els clients de diferents maneres. Un altre requisit en aquest sentit és que el *GameObject* ha d'heretar de ***NetworkBehaviour***, en comptes de l'habitual *MonoBehaviour*.

Un cop arribem a aquest punt, la manera més simple de replicar és anotar cada variable del *GameObject* a sincronitzar mitjançant l'atribut **[*SyncVar*]**. Aquest mecanisme funciona amb tots els tipus proporcionats per Unity. Per tant, qualsevol variable d'aquest grup de *SyncVars*, en rebre un nou valor, el replica automàticament a tots els clients.

És per això que, quan s'utilitza una estructura pròpia com a variable, cal tenir en compte que tot el seu contingut s'envia alhora. Per tant, si és molt gran i només canvia una petita part d'ella, utilitzarem una amplada de banda excessiva.

Si necessitem utilitzar un sistema de control més refinat a l'hora de replicar variables, podem sobreescriure les funcions *OnSerialize()* i *OnDeserialize()* per llegir i escriure manualment aquestes variables. Això equival a afegir una funcionalitat personalitzada, però no es permet el seu ús combinat amb *SyncVars* i cal triar un dels dos mètodes.

Unity també implementa el mecanisme de crides remotes. En concret, Unity proporciona un mecanisme anomenat ***Command***, que representa una acció enviada des d'un client al servidor, restringit als objectes controlats pel jugador. En canvi, una funció de tipus **Client RPC** és una acció enviada des del servidor al client.

Nota

Com amb SyncVar, aquests tipus de funcions RPC només són suportades per subclasses de *NetworkBehaviour*.

El sistema per activar els mecanismes amb aquestes funcions especials és molt similar al mecanisme de sincronització de variables, requerint l'atribut **[*Command*]** i el prefix *Cmd* en cada funció, com per exemple *CmdFire*(..). I de manera anàloga, anotant amb l'atribut **[*ClientRpc*]** i nomenant amb el prefix *Rpc* activem l'altre tipus de crida remota. D'aquesta manera, qualsevol d'aquestes funcions anomenades de nou i anotades pot ser cridada per una funció local i aquesta pot ser automàticament executada de forma remota.

5.1.2. UNet LLAPI

L'API de la capa de transport proporcionada per UNet és una façana d'accés als *sockets*. Lògicament, aquesta capa de baix nivell pot ser vista com una pila de protocols de xarxa incorporada a la part superior de la capa de transport, que conté una capa de xarxa i una capa de transport.

LLAPI

El disseny de la LLAPI s'assembla molt a l'API de *sockets* BSD.

Per a la capa de transport, en comptes de sol·licitar específicament una connexió UDP o TCP, s'ha d'especificar com es vol utilitzar la connexió. És a dir, es pot crear un canal de comunicació i sol·licitar diferents valors relacionats amb la qualitat del servei (QoS):

- ***Unreliable***. Envia missatges sense cap garantia de recepció.

- ***UnreliableSequenced***. Envia missatges sense garantir l'arribada, però els missatges fora de l'ordre són descartats. Especialment útil en comunicació de veu.

- ***Reliable***. Es garanteix la recepció dels missatges, sempre que es mantingui la connexió.

- ***ReliableFragmented***. Un missatge fiable que es pot fragmentar en diversos paquets. Això és útil per enviar fitxers grans a través de la xarxa, que s'han de tornar a muntar a la destinació.

La capa de xarxa s'utilitza per crear connexions entre màquines, lliurament de paquets, i control de flux i congestió.

Les connexions es poden establir a través de la crida de funció ***NetworkTransport.Connect***. Aquesta crida retorna un identificador de connexió, que es pot utilitzar com a paràmetre en altres funcions de la classe ***NetworkTransport*** per enviar i rebre paquets, així com desconnectar-se.

Com amb els *sockets,* la LLAPI només suporta una abstracció per a l'intercanvi de missatges binaris sense format. No proporciona mecanismes d'alt nivell com serialització, codificació, crides RPC, etc.

5.2. Alternatives a UNet

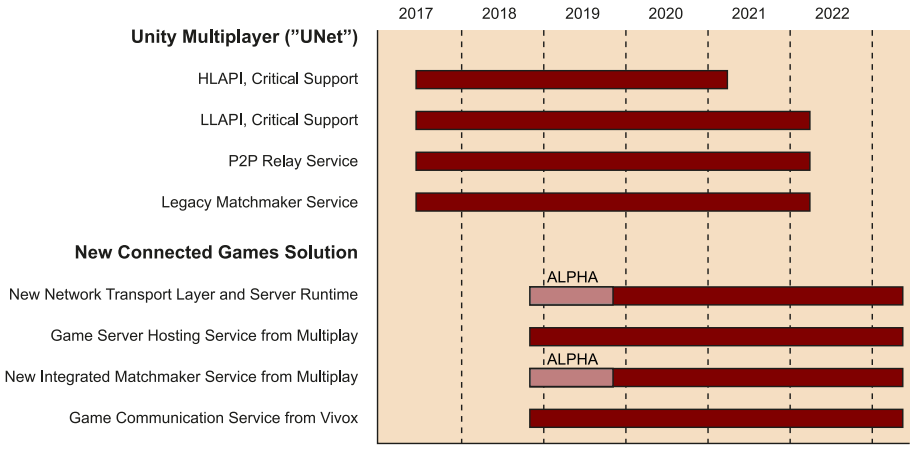
El 2018 Unity va decidir que deixava com a obsoleta (*deprecated*) la biblioteca d'UNet perquè no complia les necessitats mínimes de molts dels creadors de videojocs multijugador actuals. Aquestes necessitats bàsicament es tradueixen en fiabilitat, rendiment, escalabilitat i seguretat; característiques que UNet va deixar de complir amb el pas dels anys.

Deprecation

El fet de desaconsellar l'ús d'algunes característiques típicament perquè han estat substituïdes o ja no es consideren eficients o segures, sense eliminar-les completament per ser discontinuades en el futur.

Tanmateix, l'alternativa a les API d'UNet s'ha endarrerit bastant i, tot i que les antigues API tenen suport fins a les versions 2018.4 (LTS) en el cas d'HLAPI i 2019.4 (LTS) en el cas de LLAPI, les noves versions de la nova API de xarxa d'Unity encara no s'han alliberat.

Figura 6. *Deprecation plan* per a UNet



Mentrestant, molts desenvolupadors de jocs multijugador en Unity han optat per utilitzar altres solucions, com per exemple ***Mirror***, que proporciona una API molt similar a l'HLAPI d'Unet, resol els errors que tenia i a més està preparada per a jocs multijugador massius en línia (MMO). Aquesta llibreria està molt estesa dins la comunitat i, en ser una API d'alt nivell com HLAPI, se centra en facilitar el desenvolupament de videojocs multijugador i no en la gestió del transport, pròpiament. Així mateix, compta amb una comunitat molt activa i és un projecte de codi obert, que assegura que el suport estigui garantit.

6. Projecte: *Tanks!* LAN

Com a conclusió principal de tot el que hem vist fins ara, podem dir que el multijugador en xarxa és inherentment complex i exigeix tenir en compte molts detalls que afecten directament i indirectament els videojocs, així com problemes i dificultats particulars associades amb la sincronització i la comunicació entre diverses instàncies d'un videojoc, que sovint s'executen en diferents màquines les quals podrien trobar-se en llocs del món distants entre si.

En aquest projecte continuarem amb l'exemple del joc de tancs, en què cada jugador controla un tanc i dispara projectils per acabar amb els seus rivals, i que ja hem treballat com a multijugador local. Tanmateix, si seguim la norma general, per desenvolupar un multijugador en xarxa hem de començar de nou, òbviament a partir dels recursos (*assets*) bàsics ja disponibles (*scripts* de control, models, *shaders,* etc.), però tornant a implementar bona part de la lògica del joc amb els mecanismes de xarxa necessaris.

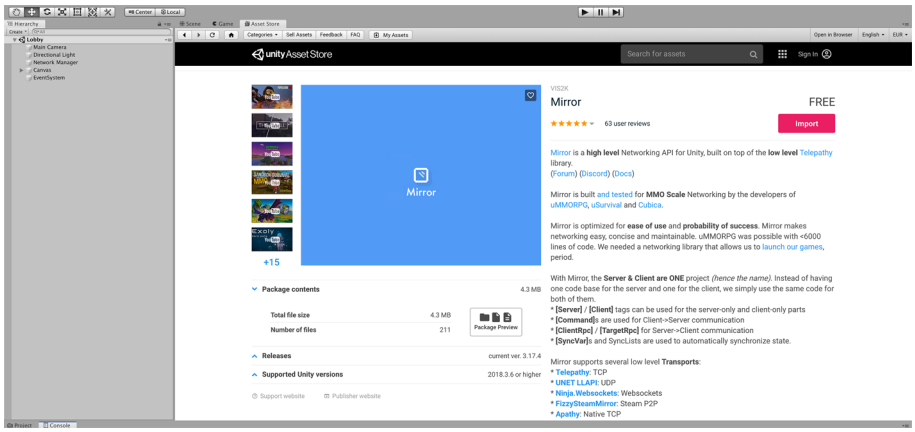
Amb aquest projecte mirarem de treballar de la manera més senzilla possible tots els mecanismes i els aspectes que permet l'API ***Mirror***. A través d'ella, detallarem pas a pas el seu funcionament i plantejarem una sèrie de reptes per millorar el joc inicial. Encara que es tracti d'un *shooter* multijugador (un dels gèneres més populars), l'ús de cada mecanisme es pot extrapolar a qualsevol gènere i a moltes situacions diverses però comunes a la resta de jocs multijugador.

Per tant, serà un projecte client/servidor al qual s'hi podran unir diversos jugadors (tancs) en xarxa i disparar projectils que afectaran la seva salut segons els impactes rebuts. Veurem com implementar el disparament de projectils amb *spawnable prefabs,* l'ús d'un *authoritative server* per controlar la salut dels diferents jugadors i com implementar les rutines de mort i reaparició mitjançant crides RPC.

6.1. Instal·lació de *Mirror*

El primer pas serà obtenir l'*asset* de ***Mirror*** de l'*asset store* d'Unity i importar-lo al nostre projecte. Un cop fet això, és imprescindible que reinicieu Unity perquè els components del menú s'actualitzin correctament.

Figura 7. Importació de l'*asset* ***Mirror*** al projecte

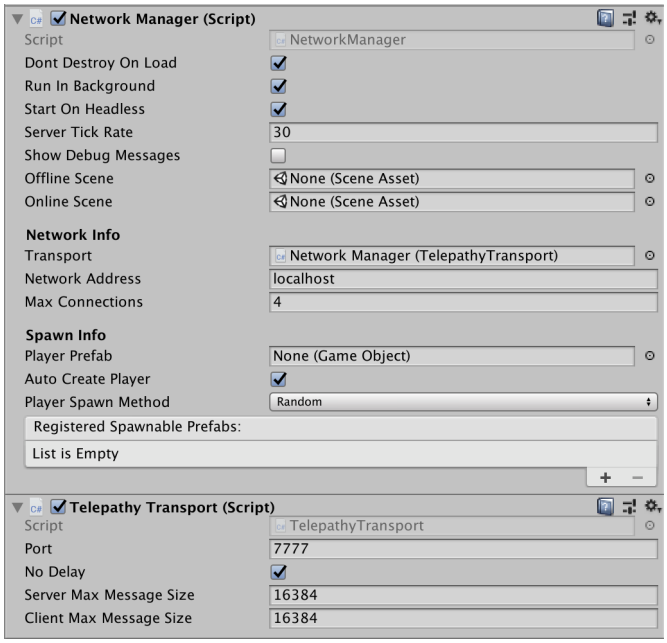


6.2. Creació de l'administrador de xarxa

Hem de crear un nou administrador de la xarxa (classe *NetworkManager*). Aquest administrador de xarxa controlarà l'estat del joc multijugador, incloent la gestió de l'estat del joc, de l'*spawn*, la gestió d'escenes i permetent l'accés a la informació de *debug*.

Per crear un nou administrador de la xarxa, n'hi ha prou d'afegir un component de tipus *NetworkManager* a un *GameObject* anomenat, per exemple, *Network Manager*. Tot seguit, podem veure una captura de pantalla del component *NetworkManager* i fer-nos una idea ràpida de com gestiona l'estat de la xarxa del joc.

Figura 8. *Script Network Manager*



Com podem observar, aquest component permet configurar-se de la següent manera:

- Evitar la seva destrucció en carregar una escena.

- Executar-se en segon pla.
- Executar-se en mode *headless* si el dispositiu no disposa de capacitats gràfiques.
- Freqüència (*tick rate*) amb què un servidor actualitza l'estat del joc. Només s'utilitza en mode *headless*.
- Mostra o no missatges de depuració.
- Referenciar l'escena de *lobby* associada al joc.
- Referencia l'escena de joc associada al joc.
- Protocol de transport a utilitzar (per defecte, *Mirror* utilitza les llibreries *Telepathy*), adreça de xarxa i nombre màxim de clients per servidor.
- Informació sobre la generació d'objectes de *spawn*.
- Configuració avançada del protocol de transport *Telepathy*.

6.3. Creació del jugador

L'actiu prefabricat (*prefab*) Player representarà el tanc del jugador, o jugadors. De manera predeterminada, el component *NetworkManager* instancia un *GameObject* per a cada jugador que es connecta a la xarxa, mitjançant la clonació i l'*spawn* del *prefab* Player en el joc. El *GameObject* del jugador en l'escena hauria de tenir l'aspecte següent.

Figura 9. Aspecte de l'objecte *GameObject* del jugador

Per identificar el jugador com un *GameObject* únic en el joc en xarxa, hem d'afegir un component de ***NetworkIdentity*** en el *prefab* del jugador (*Player*).

Figura 10. *Network Identity*

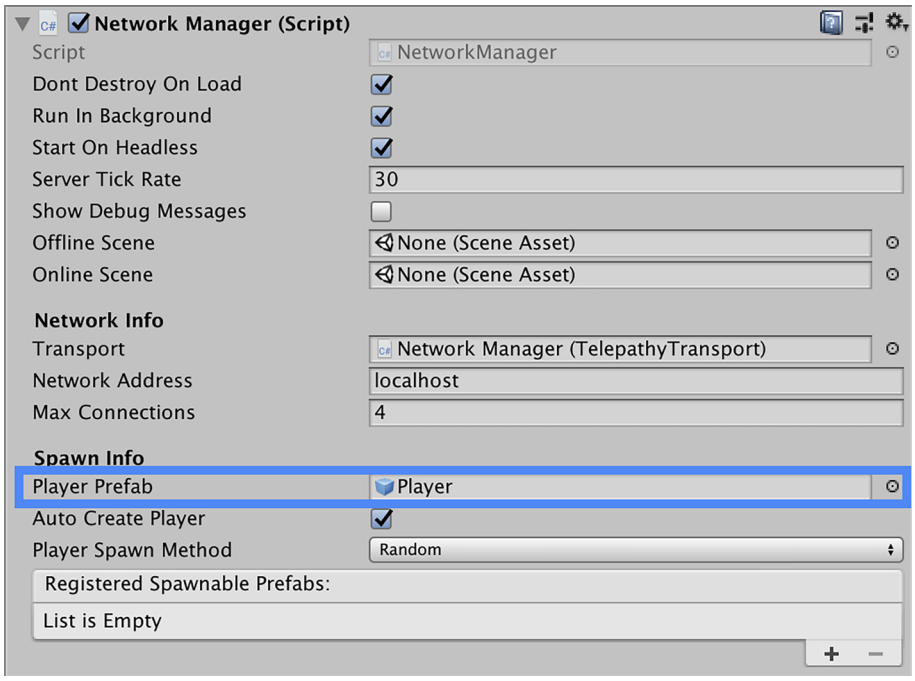
El component *NetworkIdentity* s'utilitza per identificar l'objecte a la xarxa i fer que el sistema de xarxa en sigui conscient. En aquest component, hem de tenir la propietat Local Player Authority activada per habilitar el control del moviment del *GameObject* del tanc al jugador client.

Vegeu també

Per obtenir més informació sobre el component NetworkIdentity, podeu consultar l'API mirror.

Un cop tenim preparat el *prefab* Player, aquest ha de ser registrat en el sistema de Networking. Per a això, i com que la majoria dels jocs tenen un únic i idèntic *prefab* que representa qualsevol dels jugadors en el joc, en el nostre cas hem d'indicar el *prefab* Player en la propietat del *NetworkManager* específica per a això (*Player Prefab*).

Figura 11. *Network Manager*



Player

Network Manager utilitzarà aquest *prefab* per fer *spawn* de nous *GameObjects* del jugador en l'escena.

El *GameObject* que representa el jugador es crea sempre que un nou jugador client s'uneix a la partida de l'amfitrió. En aquest punt sorgeixen altres temes molt relacionats amb l'*spawning* per xarxa i altres detalls de la sincronització de *GameObjects* de jugador entre clients i servidor. Però no els tractarem fins més endavant, un cop hàgim realitzat la resta dels passos bàsics.

6.4. Creació del controlador del jugador

Control de l'entrada

Per defecte, *Input.GetAxis("Horitzontal")* i *Input.GetAxis("Vertical")* permeten al jugador moure's amb les tecles WASD o les tecles de direcció, un *gamepad* o un altre dispositiu d'entrada.

Un cop tenim les bases per a la gestió dels objectes en el joc en xarxa, ens centrarem en els aspectes relacionats amb la mecànica del joc en si. Un primer pas és poder moure el *GameObject* del tanc jugador per l'escena. Per a això comptem amb l'*script TankController*, que seria el següent sense cap codi de xarxa.

```
using UnityEngine;


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


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

Com veiem, es tracta del codi típic del mètode *Update()*, on s'apliquen les rotacions i les translacions del jugador local que controla el tanc. Per afegir la funcionalitat de xarxa al moviment del jugador i assegurar que només el jugador local pot controlar el *GameObject* local, hem de tenir en compte les següents premisses:

- Utilizar el *namespace Mirror*.

```
using Mirror;
```

Namespace Mirror

El *namespace Mirror* proporciona tot el codi de xarxa que necessitem per escriure els nostres *scripts*.

- *Estendre* el *NetworkBehaviour*, en comptes de *MonoBehaviour*.

```
public class TankController : NetworkBehaviour
```

NetworkBehaviour

Els *scripts* afegits en un *GameObject* que necessiten funcions de xarxa han d'heretar de *NetworkBehaviour*, que al seu torn estén *MonoBehaviour*.

- S'ha d'afegir una comprovació de la propietat *isLocalPlayer* en la funció *Update*, perquè la realitzi només el jugador local.

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

Aquest punt es deu al fet que, en un projecte en xarxa, el servidor i tots els clients estan executant el mateix codi amb les mateixes seqüències de comandaments en els mateixos *GameObjects* alhora. A més, tots aquests *GameObjects* jugador es van generar a partir del mateix *prefab* Player i tots ells executen el mateix *script TankController*.

Vegeu també

La classe base *NetworkBehaviour* proporciona altres funcionalitats molt útils. Per a més informació, consulteu l'API oficial de *Mirror*.

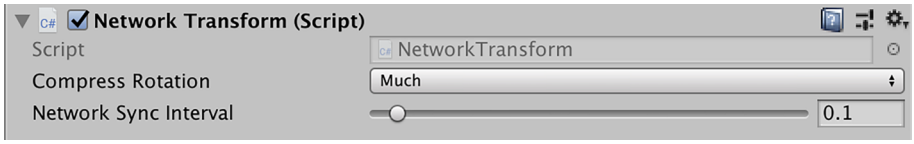
Precisament per evitar que la seqüència sigui exactament igual, fem ús del concepte *LocalPlayer*, que fa referència al *GameObject* del jugador en propietat del client local.

La propietat *isLocalPlayer* s'estableix automàticament a través del *NetworkManager* sempre que un client es connecta al servidor i nous *GameObjects* jugador es creen a partir del *prefab* Player. Quan un client es connecta al servidor, la instància creada en el client local es marca com a *LocalPlayer*. Tots els altres *GameObjects* que representen aquest jugador (al servidor o en un altre client) no estan marcats amb aquesta propietat.

Si executem el projecte en aquest punt, els *GameObjects* jugador no es mantenen sincronitzats a través de totes les instàncies del projecte en curs. Els controls d'entrada per al moviment només es processen en el jugador *GameObject* local, però la posició i la rotació d'aquest jugador no s'actualitzaran en la resta de les instàncies del joc.

Per mantenir els *GameObjects* dels jugadors en sincronia, necessitem afegir un component anomenat ***NetworkTransform*** al *prefab* Player. Aquest component precisament s'encarregarà de sincronitzar el component Transform d'un *GameObject* en tota la xarxa.

Figura 12. *Network Transform*



NetworkManagerHUD

El component *NetworkManagerHUD* proporciona una interfície d'usuari senzilla per gestionar la partida en xarxa.

Repte 1

Afegiu un mode temporal per inicialitzar fàcilment el joc en mode servidor i en mode client a partir de l'escena *Main*.

Concretament, la propietat ***Network Sync Interval*** del *NetworkTransform* permet ajustar la ràtio de fluïdesa en què se sincronitzarà la localització d'aquest *GameObject* en concret, especificant cada quan se sincronitza (en segons).

Repte 2

Realitzeu una sèrie de proves amb diferents configuracions de *NetworkTransform* i analitzeu els resultats obtinguts.

6.5. Identificació del jugador

Inicialment, tots els *GameObjects* que representen els jugadors tenen les mateixes característiques i són idèntics a simple vista. Per tant, perquè el jugador local pugui identificar quin és el seu tanc, hem de fer algun canvi que sigui fàcilment visible a la pantalla.

La funció *OnStartLocalPlayer* és un bon lloc per fer la inicialització del jugador local com, per exemple, la configuració de les càmeres de la configuració i els controls d'entrada. Aquesta funció només és cridada pel *LocalPlayer* en el seu client. En aquest cas, veiem en la implementació la funció *OnStartLocalPlayer*:

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

Això farà que cada jugador pinti els materials del *prefab* Player de color blau i que cada jugador vegi d'aquest color el tanc que controla en la seva instància del joc.

Figura 13. Resultat de l'aplicació del codi

6.6. Disparament de projectils

Una característica comuna en els jocs multijugador és que les accions de cada jugador tenen efecte sobre la resta dels jugadors. En aquest cas, es tracta de disparar projectils amb l'objectiu de derrotar l'adversari. Per fer-ho, caldrà revisar *TankController* perquè cridi una funció que s'encarregui de realitzar aquesta acció remota a través de la xarxa.

- Afegir una referència al *prefab* del projectil i una altra al lloc on es realitzarà l'*spawn*.

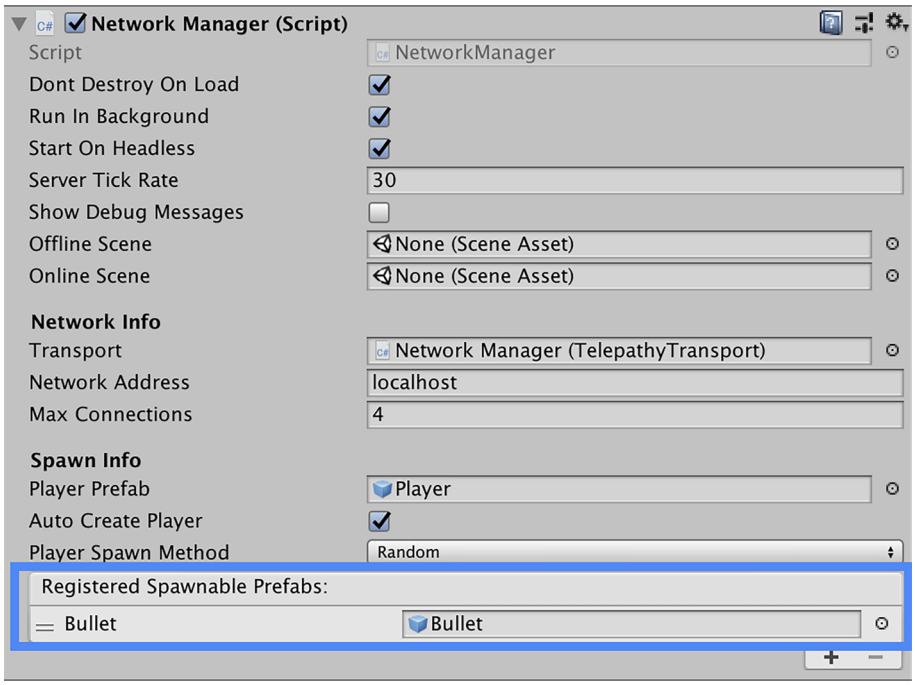
```
public GameObject bulletPrefab;
public Transform bulletSpawn;
```

- A la funció *Update()*, afegiu controls d'entrada per a l'acció de disparar.

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

Altres factors importants a tenir en compte que ja hem vist en apartats anteriors són que el *prefab* del projectil necessita una ***NetworkIdentity*** per identificar-lo com a únic a la xarxa, una ***NetworkTransform*** per sincronitzar la posició i la rotació del projectil, i que ha de ser registrat en el ***NetworkManager*** com a *prefab* ***spawnable***.

Figura 14. Registre del *Bullet Prefab Spawnable*



Tanmateix, com que el projectil no canvia de direcció o velocitat després d'haver estat disparat, no cal enviar actualitzacions dels moviments. Cada client pot calcular de manera fiable la posició del projectil un cop sap la seva posició inicial. És per això que, una optimització senzilla, consisteix únicament a no afegir-hi el component *NetworkTransform*, de manera que la posició no se sincronitzi a través de la xarxa i així es permeti a cada client calcular la posició del projectil per si mateix, tot reduint el trànsit global de la xarxa i millorant el rendiment del joc.

A continuació, l'*script TankController* també ha d'estar preparat per treballar en xarxa amb la funció *CmdFire*. Però primer recapitulem els passos anteriors, ja que com hem vist, el principi bàsic és que el servidor i tots els clients estan executant el mateix codi en els mateixos *GameObjects* al mateix temps. A més, hem vist una manera de controlar aquestes execucions mitjançant la propietat *isLocalPlayer*.

Però tenim una altra forma de control més potent anomenada *Command*, que és cridada pel client però executada en el servidor. Per utilitzar aquesta nova crida remota, hem de marcar la funció amb l'atribut [*Command*]. Qualsevol argument en la funció s'envia automàticament al servidor en la mateixa crida com a paràmetre remot.

Commands

Els *Commands* només es poden enviar des de l'objecte del jugador local. En declarar una funció com a *Command*, el seu nom ha de començar amb el prefix *Cmd*.

Seguint aquestes indicacions, al l'*script TaskController* tenim l'atribut [*Command*] per a la funció *CmdFire*:

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

Un altre concepte únic en aquest escenari és el concepte de *Network Spawning*, pel qual el mecanisme de *spawn* implica **quelcom més que generar una instància**. En concret, implica crear un *GameObject* al servidor i en tots els clients connectats al servidor. El *GameObject* llavors serà gestionat pel sistema de *spawn* i les seves actualitzacions d'estat s'envien als clients quan canvia en el servidor. Quan la seva referència es destrueix al servidor, els *GameObjects* també seran destruïts en els clients.

Vegeu també

Per obtenir més informació sobre l'atribut [*Command*], reviseu la documentació oficial.

Per crear un objecte mitjançant aquest sistema, s'utilitza la funció *NetworkServer.Spawn(..)*. Per exemple, per crear el projectil es faria de la següent manera:

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

Finalment, el resultat del *TankController* amb aquests conceptes incorporats és:

```
            [Command]
            void CmdFire() {

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

                NetworkServer.Spawn (bullet);

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

El primer pas per provocar un projectil afecti el seu objectiu és afegir lògica de control de col·lisions. Per a aquest exemple, simplement es destrueix el *GameObject* del projectil quan aquest colpeja qualsevol altre objecte amb un *Collider* associat.

```
        using UnityEngine;

        public class Bullet : MonoBehaviour {

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

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

Veiem que el projectil serà instanciat en la posició corresponent i des d'aquí definim la velocitat desitjada. Com que aquest projectil és gestionat per *NetworkManager*, quan es destrueix en el servidor també serà automàticament destruït en tots els clients, fent consistents totes les instàncies.

Repte 3

Afegiu altres tancs que no estiguin controlats per cap jugador i que es vegin afectats pels projectils que disparen els jugadors.

6.7. Control de la salut del tanc

Pel que fa a la sincronització dels estats dels jugadors, sabem que els projectils causen danys en impactar, tot afectant la propietat que representa la salut dels tancs. Per tant, també cal sincronitzar l'estat de salut de cada tanc en totes les instàncies del joc.

Primer veurem com es gestiona la salut del jugador, a través de l'*script Health,* el qual gestiona el valor de vida actual i té declarat el seu valor màxim:

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

La lògica de control dels danys rebuts i la conseqüent reducció de salut o mort del jugador després d'un impacte determinat és la següent:

```
public void TakeDamage (int amount) {

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

Com que els impactes en aquest joc només són produïts pels projectils, també hem de considerar l'*script Bullet*, que és el que crida la funció *TakeDamage* durant l'execució de la funció *OnCollisionEnter*, que ja hem vist i que una vegada completada queda de la següent manera:

```
void OnCollisionEnter (Collision collision) {

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

```
        }

        Destroy(gameObject);
    }
```

En aquest projecte proporcionem un *GameObject* per a la barra de vida (*HealthBar*) amb el seu component de *script* (*Health*), de manera que permet visualitzar la vida actual dels tancs com succeeix en molts jocs, com per exemple en la saga *Command & Conquer*.

L'estratègia que seguirem en aquest cas és la ***authoritative server***, segons la qual els canvis en la salut jugador són controlats per la instància del servidor. Aquests canvis són posteriorment sincronitzats pel client. Per aplicar-la, hem d'utilitzar el mecanisme de sincronització d'estat que ens facilita l'API *Mirror*.

Aquest mecanisme s'habilita utilitzant un altre atribut, en concret [*SyncVar*], que converteix les propietats dels *GameObjects* en atributs de tipus *SyncVars*. A continuació, detallarem com queda l'*script* de *Health* amb aquest mecanisme configurat.

Hem d'utilitzar el *namespace* d'*UnityEngine.Networking,* estendre la classe *NetworkBehaviour* i configurar la propietat *currentHealth* com a *SyncVar*:

```
        using Mirror;

        public class Health : NetworkBehaviour {

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

També hem d'afegir una comprovació en la funció *TakeDamage* perquè només sigui el servidor el que aplica el canvi de dany.

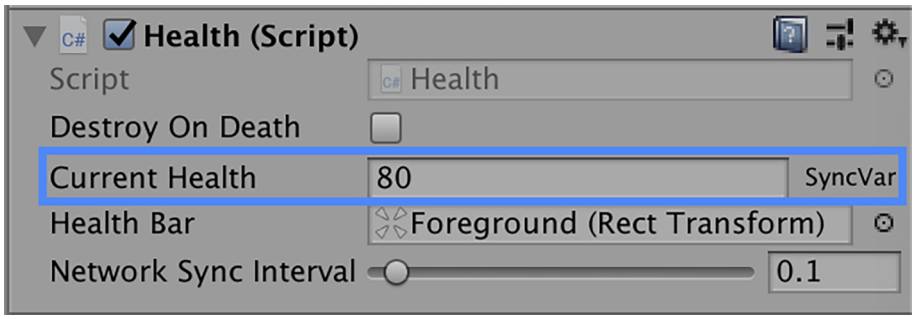
```
        if (!isServer)  {
            return;
        }
```

Així és com la salut del jugador només es modifica en el servidor i és posteriorment sincronitzada en tots els clients.

Tanmateix, en aquest punt del projecte, el *GameObject* de *HealthBar* no funciona correctament en totes les instàncies del joc, sense visualitzar-se en els clients tal com ho hauria de fer.

Si ens centrem en la variable *currentHealth*, veiem que és pública i es pot veure en l'editor. Si l'editor s'està executant en mode client, no com a servidor, utilitzant l'inspector podem veure la salut actual del jugador.

Figura 15. *Current Health*



També podem veure funcionant les barres de salut a l'escena, però només en local, en el client actual i no en els altres clients, com hauria de ser. Això és perquè no estem sincronitzant el valor de *RectTransform* a través de la xarxa ni executant el codi per ajustar la mida de la *HealthBar* en els altres clients, i només s'està executant en el client local i el servidor.

HealthBar en local

El motiu pel qual la *HealthBar* només treballa en el *host* client és perquè és local en el servidor, tot compartint la mateixa escena.

Per tant, el següent pas és sincronitzar el *RectTransform* de la *HealthBar*. Per fer-ho, utilitzarem una altra eina de sincronització d'estat: el *SyncVar hook*. Els *SyncVar hooks* enllacen amb una funció *SyncVar*. Aquestes funcions s'invoquen en el servidor i tots els clients s'actualitzen quan canvia el valor del *SyncVar*.

Vegeu també

Per a més informació sobre els mecanismes de *SyncVars* podeu revisar la documentació oficial sobre sincronització de l'estat de *Mirror*.

Per començar, els canvis en la *HealthBar* es produeixen en la funció *OnChangeHealth(..)*.

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

Convé assenyalar que aquesta funció ha de tenir un paràmetre del mateix tipus que la variable amb l'atribut [*SyncVar*], en aquest cas *currentHealth*, i que el valor actual de *SyncVar* s'enviarà a la funció amb *SyncVar hook* com a paràmetre adjunt. Per establir un enllaç a aquesta nova funció en l'atribut *SyncVar* per a *currentHealth*, hem de declarar-ho de la següent manera:

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

Ara sí, quan el valor de la propietat *currentHealth* canviï, *OnChangedHealth(..)* serà cridat al servidor i en tots els clients per actualitzar la *HealthBar*. I així és com aconseguim que la *HealthBar* se sincronitzi en totes les instàncies del joc. Per tant, quan els jugadors es disparen entre ells, totes les barres de salut han de reflectir el valor de la salut actual del jugador.

Repte 4

Quins mecanismes hem d'utilitzar per implementar la funcionalitat del carregador? Penseu a resoldre tant la gestió del nombre de projectils com la manera de mostrar aquesta informació.

6.8. Mort i reaparició

Fins a aquest punt, no passa res si la salut del jugador arriba a zero, a excepció d'un missatge en la consola del servidor. Per finalitzar el *gameplay* d'aquest joc, hem de revisar les accions de mort i *respawn* del jugador, és a dir, quan la salut arriba a zero, aquest ha de desaparèixer i tot seguit tornar a aparèixer en un altre punt en el mapa amb la salut restaurada.

Aquest últim pas ens permet introduir un altre mecanisme per a la sincronització d'estat: les crides ***ClientRpc***. Aquestes crides remotes permeten enviar des de qualsevol *GameObject* generat en el servidor amb un component *NetworkIdentity*.

Nota

Tot i que una funció *ClientRpc* és cridada en el servidor, en realitat serà executada en els clients.

Per tant, podem afirmar que les crides *ClientRpc* són el mecanisme invers a un *Command*, ja que com hem vist anteriorment, aquests són cridats en el client, però són executats en el servidor. També, de manera semblant a *Command*, per convertir una funció en una crida *ClientRpc*, necessitem afegir l'atribut [*ClientRpc*] i el prefix *Rpc* al nom de la funció. A més, qualsevol argument es passarà automàticament com a paràmetre als clients com a part de la crida *ClientRpc*.

Vegeu també

Per a més informació sobre les crides *ClientRpc*, podeu revisar la documentació oficial de *Mirror*.

Per tant, per habilitar el mecanisme de *respawning* en el joc necessitem la funció de *Respawn* amb el prefix *Rpc* i l'atribut [*ClientRpc*] dins de l'*script Health*.

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

La crida es fa en la funció *TakeDamage* del servidor, quan la salut actual del jugador arriba a zero. A més, el codi de *TakeDamage* ha de restablir la salut actual del jugador al màxim.

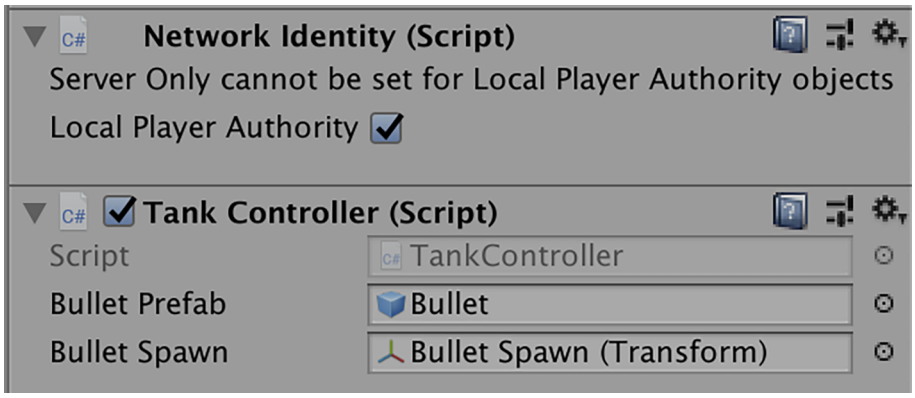
```
        currentHealth = maxHealth;
        RpcRespawn();
```

Fins ara, el client controla la posició del jugador *GameObject* local, ja que té activada la propietat ***Local Player Authority*** permesa en *NetworkIdentity* del *prefab* del tanc del jugador.

Nota

Recordeu que *RpcRespawn()* es cridarà en el servidor, però s'invocarà en els clients.

Figura 16. Propietat *Local Player Authority*



Per això, quan la vida arriba a zero, el servidor simplement restaura la posició del *GameObject* del tanc a l'origen i el client tindria l'autoritat per aturar aquesta acció.

Per evitar que això passi, el servidor procedeix de manera que explícitament ordena al client que mogui el tanc del jugador a la posició de reinici mitjançant una crida *ClientRpc*. D'aquesta manera, aquesta posició se sincronitza a través de tots els clients gràcies al component *NetworkTransform* del *GameObject* del jugador.

Podeu comprovar que funciona correctament quan un jugador dispara a un altre i li fa prou danys com per deixar la seva salut a zero; llavors, el *GameObject* del tanc danyat desapareix, torna a l'origen i la seva salut actual es restaura al màxim.

Repte 5

Destruïu els enemics NPC en quedar-se amb la salut a zero després de ser impactats per projectils dels jugadors.

6.9. *Lobby*

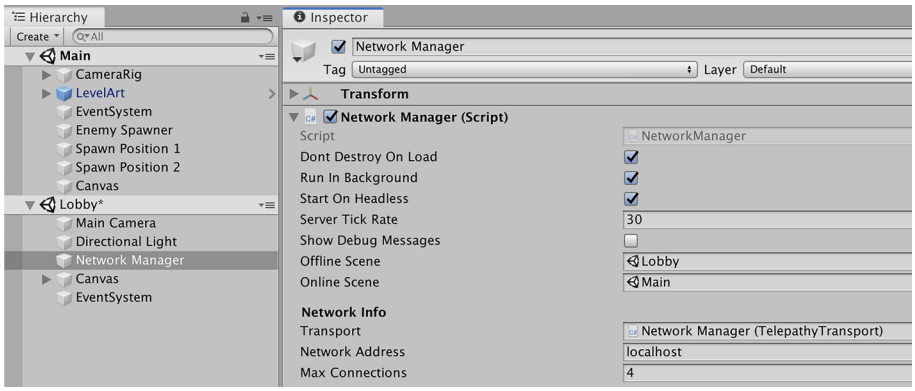
Finalment, per completar el projecte, parlarem de l'escena *Lobby* que ens permet crear una partida com a *host* o unir-nos a altres partides ja creades.

Figura 17. Escena *Lobby*

Per fer-ho, el primer pas és crear l'escena *Lobby* i moure aquí el *GameObject Network Manager*. Com que crearem un menú amb la interfície d'usuari d'Unity, necessitarem el *GameObject* de tipus Event System perquè reculli les interaccions d'usuari.

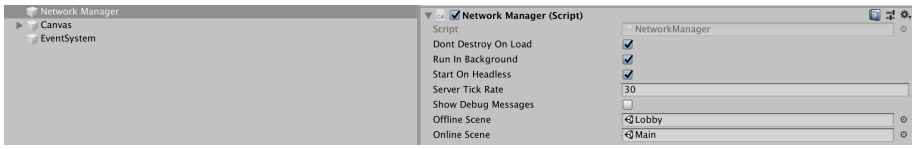
La càrrega de les dues escenes al mateix temps fa que sigui fàcil per a nosaltres moure o copiar *GameObjects* d'un a un altre.

Figura 18. El *Network Manager* de l'escena *Lobby*



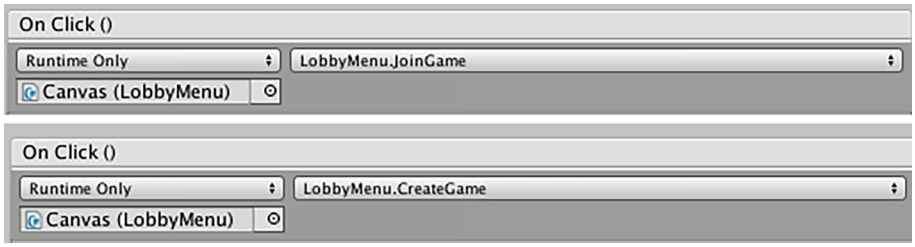
Amb l'escena *Lobby* i l'*script LobbyMenu* veurem implementada la manera més simple de començar la partida multijugador en xarxa local com a *host*. En primer lloc, indiquem a *NetworkManager* que quan no estigui connectat ensenyi l'escena *Lobby* i que quan ho estigui ensenyi l'escena *Main*.

Figura 19. Configuració d'escenes *Lobby-Main* en el *Network Manager*



A continuació, necessitem la interfície gràfica per interactuar; a partir del *canvas,* afegim dos botons similars que ens permetran cridar les accions per crear una partida i una altra per a unir-nos-hi.

Figura 20. Configuració de botons de crida a l'acció



Aquestes funcions requereixen les funcions de xarxa necessàries tant per gestionar la partida (*NetworkManager*) tot especificant la IP del servidor perquè els clients puguin connectar-s'hi. Com veiem, *LobbyMenu* s'implementa de la següent manera:

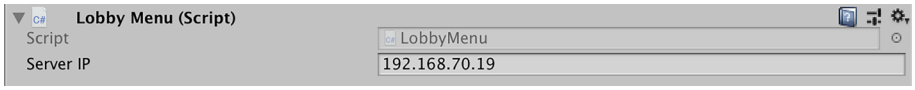
```
using UnityEngine;
using Mirror;

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 les versions de l' HLAPI d'UNet disposàvem de funcionalitats per fer *Discovery* de la xarxa tot executant un *broadcast* d'una manera totalment abstreta i transparent per a nosaltres. *Mirror* no proporciona cap funcionalitat d'aquesta mena i això ens obliga a tenir configurada de manera predeterminada la IP del servidor que allotjarà les partides.

En qualsevol cas, mostrem aquest codi a efectes demostratius. Òbviament, l'IP del servidor s'hauria de poder configurar si volem permetre el joc en LAN o utilitzar un servei de *matchmaking*, com veurem en el proper mòdul.

Figura 21. *Lobby Menu*



En l'*script Lobby Menu* configurem la IP del servidor si volem provar el nostre joc en LAN, o *localhost* si volem provar el nostre joc localment.

En el punt del projecte en xarxa, el servidor i tots els clients s'executen en un escenari on el servidor fa al mateix temps de jugador i de *host.* Si volguéssim un escenari amb un servidor dedicat exclusivament, hauríem de tenir tres instàncies del joc per a dos jugadors: un servidor i dos clients, amb dos *GameObjects* per als jugadors en cadascun, per a un total de sis diferents *GameObjects* de jugadors diferents.

Repte 6

Modificar el projecte perquè que el lobby aparegui l'opció alternativa d'iniciar una sessió de servidor dedicat.

Ja només falta, tot i que quedi fora de l'abast d'aquest projecte, mantenir i administrar aquestes connexions, així com les tasques de revisar les desconnexions i les reconnexions dels jugadors i el servidor, i que el joc sigui capaç de ser jugat contínuament.

Repte 7

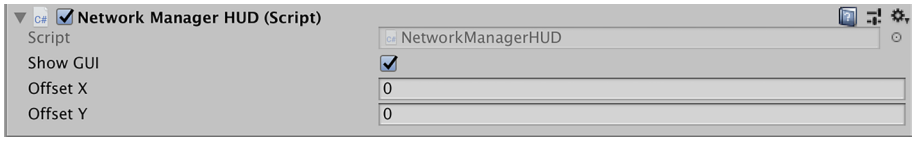
Mostreu per consola les dades de xarxa més rellevants del servidor i dels clients connectats.

6.10. Solució als reptes proposats

Repte 1

El component *NetworkManagerHUD* funciona amb *NetworkManager* i proporciona una interfície d'usuari senzilla per controlar l'estat de la xarxa del joc quan aquest s'executa.

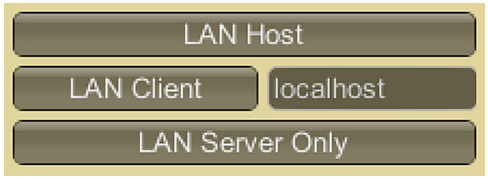
Figura 22. *Network Manager* HUD



Per provar el moviment en mode multijugador hem de tenir dues instàncies del joc executant-se simultàniament, un cop hàgim generat el joc per executar-se a la nostra màquina.

La primera instància de joc que arrenquem mostrarà la interfície gràfica *NetworkManagerHUD* que es mostra de la següent manera.

Figura 23. Interfície gràfica del *Network Manager* HUD



Per provar el multijugador:

- En aquest primer cas, hem de fer clic a *LAN Host* per crear la sessió de joc i hauríem de començar a jugar en solitari tot esperant el següent jugador.

- Arrenquem una segona instància i fem clic aquest cop a *LAN Client* per connectar amb el *host* de la primera instància.

- D'aquesta manera, hauríem de veure en ambdues instàncies els jugadors connectats i les dues pantalles haurien de mostrar la mateixa escena.

- Amb la finestra d'una de les instàncies activa i amb els comandaments del jugador, s'hauria de moure a les dues pantalles.

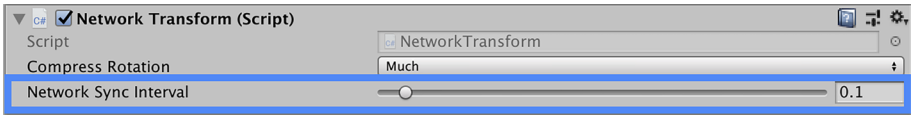
Repte 2

En aquest punt del projecte, els dos *GameObjects* del jugador han de ser completament independents entre si i estar sincronitzats amb les dues instàncies del joc, de manera simultània. Però podem notar que, segons la configuració

del *Network Sync Interval* que tinguem quan s'executen dues instàncies del projecte, el jugador remot no sembla moure's suaument en el client local, donant una sensació de **moviment a salts**.

Cal considerar en aquest punt la velocitat a què les dades es poden desplaçar entre els clients i el servidor. Per aquest motiu, *NetworkTransform* té un paràmetre de configuració que justament ens permet establir la freqüència amb què l'objecte *NetworkTransform* envia dades de sincronització per xarxa.

Figura 24. *Network Transform*



La freqüència de les actualitzacions a través d'una xarxa entre els clients i el servidor pot tenir un gran impacte en la manera com el jugador percep el joc. En aquest punt, convé considerar alguns punts clau. S'haurà d'establir un equilibri entre la freqüència amb què es produeix la sincronització de l'estat i el rendiment del joc. Si un joc prova de sincronitzar massa dades amb massa freqüència per la xarxa, el rendiment del joc patirà efectes contraproduents. Si un joc no sincronitza les dades amb prou freqüència, llavors la sensació del joc és poc fluida.

Cal tenir en compte que en tots els jocs multijugador per xarxa mai no s'aconsegueix una sincronització perfecta, ja que això és físicament impossible. Dos o més clients remots no poden tenir el mateix estat al mateix temps, a causa simplement del temps que es triga a transferir les dades entre ells. Els jugadors, tanmateix, necessiten sentir com si la seva instància estigués en sincronització per tenir la millor experiència, malgrat que en realitat les instàncies són lleugerament diferents en el seu estat exacte.

Repte 3

El nostre projecte actual s'ha centrat principalment en el jugador. La majoria dels jocs, en canvi, tenen molts *GameObjects* que no són controlats pel jugador (per exemple, NPC, *Non-Player Character*) o altres *GameObjects* de tipus ambiental, però que també coexisteixen en el món del joc.

NPC

Els personatges no controlats pel jugador són coneguts com a NPC.

En aquest repte, ens centrarem en els *GameObjects* que considerem directament enemics. Mentre que del *GameObject* del jugador es fa *spawn* quan un client es connecta al *host,* els *GameObjects* enemics necessiten ser controlats pel servidor.

Així doncs, el primer pas és crear un *GameObject* anomenat *Enemy Spawner* que s'encarregui de crear els enemics mitjançant un component *NetworkIdentity*, configurat com a *Server Only*. En activar aquesta propietat, evitarem que

l'Enemy Spawner s'enviï als clients. Per complementar aquest *GameObject*, necessitarem definir el seu comportament mitjançant un component de *script* que hem anomenat *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);
        }
    }
}
```

Aquest *script* hereta de *NetworkBehaviour*, i sobreescriu la funció *OnStartServer()*. D'aquesta manera, quan el servidor s'inicialitza, genera un nombre d'enemics amb posicions i rotacions aleatòries i finalment executa el mecanisme de *spawn* mitjançant la funció *NetworkServer.Spawn (enemy)*. Noteu que la funció *OnStartServer()* és molt similar a la que ja hem vist abans*: OnStartLocalPlayer()*. En aquest cas, *OnStartServer()* és cridat al servidor quan aquest comença a escoltar de la xarxa.

D'altra banda, per facilitar aquesta tasca de *spawn* d'enemics, farem servir el *prefab* Player com a base per al *GameObject* del tanc enemic. Aquest *prefab* també necessita els components *NetworkIdentity* i *NetworkTransform* ja inclosos, així com l'*script Health* i la configuració de la *HealthBar*. En canvi, hem d'eliminar el *GameObject* fill de *Bullet Spawn* i el component de *Tank Controller*.

Ara hem de registrar aquest *prefab* a l'*Enemy Spawner* perquè sigui utilitzat per a la creació d'enemics.

Figura 25. *Enemy Spawner*

Noteu que en no canviar el color del *prefab,* tant els jugadors enemics controlats per altres jugadors remots, com els enemics *NPC* tindran el mateix color base. A més, com que el *prefab* Enemy també té els *scripts Health* i *Bullet*, detectarà els enemics de la mateixa manera que si fossin del tipus Player.

Repte 4

Per implementar la gestió del nombre de projectils, seguirem la mateixa idea que en el cas de la salut del jugador, una propietat amb el nombre màxim de projectils del carregador i una altra amb el valor actual utilitzant el mecanisme *SyncVar*.

A l'hora de mostrar aquesta informació, podríem mantenir-la només en local, i de fet és molt habitual en aquesta mena de jocs que només el jugador vegi aquesta informació a la seva GUI. Si volguéssim donar-li un toc estratègic i que els altres jugadors també sabessin l'estatus del carregador, hauríem de seguir el mateix exemple que la barra de vida i utilitzar el mecanisme de *SyncVar hook*.

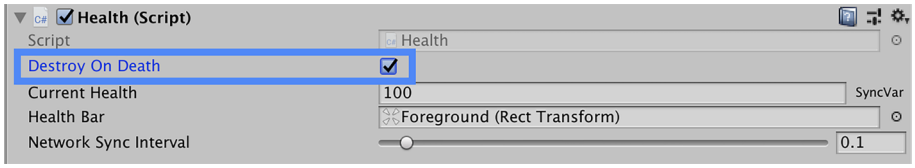
Repte 5

Si executem el projecte en aquest punt, veurem que es creen quatre enemics no controlats en crear una nova sessió de joc, que no poden ser destruïts mitjançant disparaments de projectil del nostre tanc.

Per a això, hem d'ajustar l'*script Health* perquè tingui en compte la diferència entre els tancs controlats i els no controlats. La manera més fàcil és crear una nova propietat en aquest *script*:

```
        public bool destroyOnDeath;
```

Figura 26. *Script Health*



En la funció *TakeDamage()*, hem de comprovar aquesta propietat quan la vida actual arriba a zero abans de realitzar el *respawn*:

```
        if (currentHealth <= 0) {
            if (destroyOnDeath) {
                Destroy(gameObject);
```

```
            }
            else {
                currentHealth = maxHealth;
                RpcRespawn();
            }
        }
```

Ara, l'acció de disparar els enemics farà que perdin la salut i quan aquesta arribi a zero els enemics seran destruïts. Els jugadors, tanmateix, continuen tenint el comportament anterior de tornar al punt d'origen quan moren, amb la seva salut restaurada al màxim.

Repte 6

Per poder implementar un servidor dedicat amb aquesta solució, primer hauríem de permetre que l'escena *Main* s'executi en mode servidor i no en mode *host*, com ho fèiem fins ara de del *Lobby*.

Per a això, hem d'afegir una nova funció, que anomenarem *RunServer()*, molt similar a *CreateGame()*, en l'*script* de *LobbyMenu*:

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

Una cop tinguem aquesta propietat, hem de modificar la interfície gràfica per afegir també un botó similar al *CreateButton* que cridi aquesta nova funció.

Quan tenim aquest nou mecanisme preparat, podem generar el joc de nou i arrencar dues instàncies al mateix temps per comprovar el seu funcionament.

Figura 27. Inici del joc amb dues instàncies simultànies

Repte 7

Com a suggeriment típic en aquests casos, en les primeres proves és millor començar iniciant el servidor dedicat des de l'editor d'Unity i fer *debug* tot visualitzant els rastres de *log* per comprovar que tot succeeix tal com esperem. Després podem iniciar el primer client i posteriorment el segon client per analitzar què està succeint de manera directa.

Per obtenir dades interessants, afegirem a les funcions del *LobbyMenu* que creen el *server*, *host* o client la crida a la següent funció:

```
    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());
    }
```

Per obtenir l'adreça IP de la nostra màquina ens ajudarem del mètode auxiliar *GetLocalIPAddress()* que ens la retorna:

```
    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 d'aquest punt, si necessitem més control sobre els possibles esdeveniments que succeeixen en el nostre *NetworkManager*, com per exemple reaccionar quan des del servidor es detecta que un client es connecta (*OnServerConnect*) o desconnecta (*OnServerDisconnect*), o bé quan un client detecta que el servidor ha desaparegut (*OnServerDisconnect*), hem de sobreescriure el nostre propi *NetworkManager*. Si a més requerim eines més sofisticades, llavors podem analitzar el tràfic generat pel nostre joc mitjançant *sniffers*, *profilers*, etc.

6.11. Sumari

Amb aquest projecte hem cobert la majoria dels conceptes bàsics de l'API de *Mirror*, que com hem dit està fortament basada en l'obsoleta HLAPI d'UNeT, per fer un simple joc multijugador en xarxa.

Com hem vist, quan s'utilitza *Mirror* el servidor i tots els clients estan executant el mateix codi al mateix temps en els *scripts* associats als *GameObjects*.

Hem vist com podem mantenir les transformacions dels nostres *GameObjects* en xarxa sincronitzada amb la del *NetworkIdentity* i el component *NetworkTransform*. Hem treballat amb altres mecanismes de sincronització de *SyncVars* i *SyncVar Hooks*, i ara podem mantenir el valor de les variables de manera consistent malgrat els canvis.

També hem cobert els mecanismes RPC disponibles en *Mirror*. Tant *Command* com *ClientRpc*, essent complementàries en allò referent a l'origen i el destí entre client i servidor.

Finalment, hem diferenciat les escenes de joc i de *Lobby*, així com les estratègies de client i servidor (*host*) al mateix temps o clients i servidor dedicat. Aquestes opcions s'han de tenir molt en compte a l'hora de distribuir i desplegar el joc.

Esperem que aquest projecte sigui útil com a punt de partida per entendre l'ús dels mecanismes per a videojocs multijugador en xarxa i com a visió general dels aspectes més importants de l'API d'alt nivell proporcionada per Unity.

Resum

Com hem vist, el multijugador en xarxa requereix coneixements i nocions molt avançades en temes de xarxes. No és l'objectiu d'aquest mòdul introduir aspectes avançats de programació en xarxa, però sí tenir molt clares les implicacions de les topologies i els mecanismes escollits, i els avantatges i desavantatges que poden tenir a l'hora de desenvolupar els nostres projectes.

Amb la serialització i la compressió, se'ns permet empaquetar un objecte i enviar-lo a un *host* remot, així com emmarcar aquestes dades de manera que el *host* remot pugui crear o trobar l'objecte apropiat per rebre les dades.

La replicació és un mecanisme que requereix una estratègia adequada per a cada tipus de joc i, per tant, cal considerar les seves diferents variants. En canvi, les invocacions remotes Via RPC són molt recomanables per a qualsevol tipus de joc, ja que simplifiquen el protocol de comunicació a implementar.

D'aquesta manera, hem de conèixer a fons mecanismes d'alt nivell, essent la capa d'aplicació on funcionen els jocs i, per tant, deixar l'elecció del protocol de la capa de transport per a una segona fase.

A més, hem d'estudiar com optimitzar i evitar problemes intrínsecs dels jocs multijugador, en els quals una de les solucions clàssiques segueix essent centralitzar la lògica del joc en una entitat neutral, habitualment anomenada servidor; aquesta és l'opció més coneguda entre els jugadors habituals, els anomenats *servidors dedicats*.

Finalment, hem vist com Unity, gràcies a l'API *Mirror*, ens permet desenvolupar un videojoc en multijugador. També podem entendre que un videojoc en local, encara que sigui multijugador, no pot ser migrat directament a multijugador en xarxa, sinó que se'n requereix una modificació a fons. Una pràctica habitual consisteix a realitzar primer el multijugador de xarxa i *a posteriori* el multijugador, o fins i tot el mode d'un jugador.

Bibliografia

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.

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

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

Yahyavi, Amir; Kemme, Bettina (2013). «Peer-to-peer architectures for massively multiplayer online games: A Surve». *ACM Comput. Surv.* (vol. 46, núm. 1, art. 9).

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