# Servicios en línea

PID_00243559

Rubén Mondéjar Andreu

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

# Índice

# Introducción

Después de estudiar y trabajar el mundo de los juegos multijugador, nos resta ver un paso más allá. Desde hace años, existen plataformas que nos permiten conectar nuestros juegos y disfrutar de una serie de servicios para complementar nuestros proyectos, sean o no juegos multijugador, o incluso juegos sin *gameplay* vinculado al modo en línea.

Estos servicios son muy conocidos hoy en día, y no es difícil encontrar juegos que nos ofrezcan, por ejemplo, una serie de retos y nos permitan superarlos o compartir dicho logro con nuestros amigos.

Por otra parte, es importante conocer cómo conseguimos desplegar esa lógica remota. Aquí es donde entran en juego todos los conceptos y tecnologías del lado servidor: los *backends*, los sistemas distribuidos, bases de datos, computación en la nube, etc. Veremos estos conceptos y cómo se integran con los clientes, véase nuestros videojuegos.

Para ilustrar mejor todo lo visto en este módulo trabajaremos un proyecto basado en la integración con una plataforma *cloud* real que permita añadir la funcionalidad de chat, un ejemplo clásico de comunicación bidireccional. Este pequeño ejemplo permite estudiar cómo integrar otras funcionalidades o servicios en línea para cualquier tipo de videojuego.

# Objetivos

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

1. Estudiar las plataformas de servicios en línea orientados a los videojuegos.
2. Estudiar el concepto de *cloud computing* y cómo aplicarlo a los videojuegos.
3. Programar la funcionalidad de chat en un juego multijugador local mediante el motor Unity y una plataforma real de tipo *cloud*.

# 1. Plataformas de servicios en línea

Podemos afirmar sin riesgo a equivocarnos que la mayoría de los jugadores habituales de hoy en día tienen perfiles en las plataformas más conocidas, como Steam, Xbox Live o PlayStation Network. Estas plataformas ofrecen servicios con muchas características, tanto para los jugadores como para los propios juegos, incluyendo *matchmaking*, estadísticas, logros, tablas de clasificación o partidas en la nube, entre otros.

Debido a que el uso de estos servicios en los videojuegos actuales es tan frecuente, los jugadores esperan que cada juego, incluso los de un solo jugador, se integre en uno de estos servicios de alguna manera significativa. En este apartado revisaremos los servicios más usuales e interesantes que podemos encontrar.

## 1.1. Características principales de Steam

Con tantas opciones, vale la pena considerar en qué plataforma integrar nuestros juegos. En la mayoría de los casos, la elección se hace basándose en la plataforma en la que se lanza el juego. Por ejemplo, todos los juegos de Xbox One deben estar integrados en el servicio de videojuegos de Xbox Live, aunque en este caso concreto también se nos permite integrarlo en otras plataformas con Windows 10. En este sentido, tenemos varias opciones para ordenadores basados en los sistemas operativos más conocidos, como Windows, Mac y Linux. Pero sin duda, el servicio más popular en estas plataformas hoy en día es el servicio de Valve Software, Steam. En cambio, en plataformas móviles, tenemos principalmente el Game Center para iOS y el Google Play Games para Android e iOS.

Antes de escribir cualquier código específico para un servicio de jugador, considerad cómo deseáis integrar el código en vuestro juego. Una opción rápida sería agregar directamente llamadas al código de servicio del jugador donde sea necesario. Así que en nuestro caso, llamaremos directamente a las funciones de Steamworks SDK en todos los archivos que necesiten usar el servicio de jugador. Sin embargo, esto se desaconseja por un par de razones:

**1)** En primer lugar, esto significa que cada desarrollador de su equipo puede necesitar tener un cierto nivel de familiaridad con Steamworks, porque el código que utiliza se extenderá a través de su base de código.

**2)** En segundo lugar, y lo que es más importante, esto hace que sea mucho más difícil integrar un servicio de jugador diferente en vuestro juego. Esto es particularmente una preocupación para los juegos multiplataforma, ya que, como se ha comentado, las diferentes plataformas tienen diferentes restricciones sobre las que se puede utilizar el servicio de jugador.

- **Autenticación**: se proporciona una variedad de diferentes API para administrar la autenticación y la propiedad del usuario.

- **Comunidad**: la API de la comunidad es un conjunto de funcionalidades que permiten acceder a información sobre otros jugadores, incluyendo pero estar limitado a: nombre de usuario, avatar y grupos a los que pertenece actualmente.

- **Estadísticas y logros**: proporciona un método fácil y eficaz de almacenar estadísticas de juego persistentes e información relativa a los logros.

- **Marcadores**: proporcionan un conjunto sólido de API enfocadas a que los jugadores puedan ver el *ranking* o tablas de clasificación de cada juego.

- **Partidas en línea**: Steam Cloud proporciona la forma más sencilla de sincronizar los datos de los juegos guardados en la nube, permitiendo a sus jugadores mantener su progreso en el juego sin molestias al cambiar entre dispositivos o incluso después de un desagradable bloqueo de la computadora.

- ***Matchmaking***: Steamworks proporciona un excelente conjunto de herramientas para el *matchmaking* multijugador perfecto, tanto para juegos basados en servidores como para juegos orientados al *lobby*.

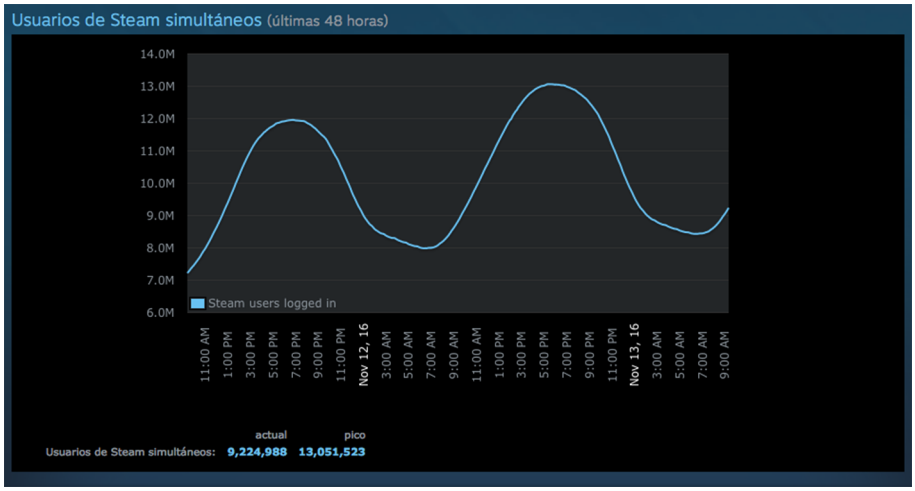
- ***Networking***: se proporciona una capa de abstracción de red para tomar la difícil logística de enviar datos a través de Internet. ¡Nunca hay que preocuparse por problemas de conectividad causados por cosas como el puerto de reenvío de nuevo!

- **Sistemas anti-trampa**: Valve Anti-Cheat está ahí para proporcionar una capa adicional de seguridad en sus competitivas experiencias multijugador. Es muy similar a un escáner de virus y tiene una lista de trucos conocidos para detectar.

Steamworks.NET fue diseñado para seguir lo más cerca posible de la API nativa Valve C++ API y cuenta con una cobertura del 100 % de la API Steamworks nativa en todas las interfaces.

## 1.2. Estadísticas

Una característica muy típica y bastante genérica de los servicios en línea es la capacidad de recopilar y mostrar diferentes estadísticas de los jugadores. De esta forma, es posible navegar por nuestro perfil o el de un amigo y ver cómo se ha completado cada juego; y plataformas como Steam o Google Play Games incluso permiten visualizar estadísticas globales, como por ejemplo el número de jugadores de cada juego durante un periodo de tiempo concreto.




Estadísticas de jugadores simultáneos en Steam

Para tener acceso a estos datos, normalmente suele haber alguna forma de consultar al servidor para ver las estadísticas del jugador, así como una forma de actualizar y escribir nuevos valores en el servidor. Aunque es viable utilizar siempre el servidor como fuente principal de datos, de forma que escribimos y leemos la información en remoto, generalmente es una buena idea almacenar en caché los valores en la memoria local del juego.

Todos estos datos generados por los jugadores durante el juego y almacenados en los servidores sirven para el análisis de juegos; y para cada juego en concreto se puede definir con flexibilidad qué datos se deben recopilar, desde los más genéricos, como horas de juego, progreso actual o puntuación total, hasta detalles más específicos de cada juego, como el porcentaje de uso de ítems, armas, tipo de enemigos eliminados, distancia recorrida, carreras ganadas, etc. De ahí se pueden extraer métricas como la frecuencia de determinados aspectos:

- Los jugadores usan un elemento en particular.
- Los jugadores alcanzan un cierto nivel.
- Los jugadores realizan alguna acción específica del juego.

De cara al desarrollador, estos datos son un gran *feedback* para saber cómo mejorar el juego. Por ejemplo, se puede ajustar el nivel de dificultad de ciertos niveles donde los jugadores se encuentran en general en situaciones extremas, ya sea porque el nivel es demasiado difícil o todo lo contrario, demasiado fácil de completar.

Además de la información individual, estas plataformas presentan diferentes tipos de estadísticas más globales, tanto públicas como privadas:

- **Estadísticas globales**: número de jugadores concurrentes, totales, picos del día en los cien juegos con más jugadores.

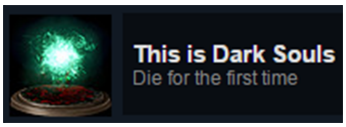
- **Estadísticas de hardware y software**: el proceso de recolección de datos sobre qué tipo de hardware y software informático están usando los jugadores, ya sea de forma automática o por medio de encuestas, ayuda a los desarrolladores a saber quiénes son realmente los jugadores.

- **Geolocalización de los jugadores**: de dónde proceden los jugadores es primordial para poder realizar una estrategia de marketing adecuada, así como planificar un lanzamiento escalonado en próximas actualizaciones o versiones del juego.

- **Compras *ingame***: saber cuánto y quién ha gastado dinero en un juego es muy importante si el sistema de monetización utilizado se basa en este tipo de estrategia comercial.

- **Informes personalizados**: este servicio también puede generar informes elaborados con toda la información recopilada de cada jugador, como sus logros más difíciles desbloqueados, total de horas jugadas, juegos más jugados, horario favorito para jugar, etc. y enviarlo de forma automática para dar un valor añadido a la plataforma a los ojos del jugador.

### 1.3. Logros (*achievements*)




**Logro**
Un logro, también conocido como *trofeo, distintivo, premio, sello, medalla* o *desafío*, es un metaobjetivo definido fuera de los parámetros de un juego.

Otra característica popular de los servicios de los jugadores son los logros. Aunque muchos otros juegos individuales desarrollan sus propios retos o secretos de forma interna, la primera implementación de un sistema de logros de fácil acceso y multijuegos fue el Xbox Live, con el *Xbox 360 Gamerscore*, introducido por Microsoft en 2005. Este sistema fue adaptado por otras plataformas y ha ido evolucionando durante estos años, aunque aún destaca por su simplicidad, y, a día de hoy, sigue siendo muy común y demandado por los jugadores habituales.

Su funcionamiento está muy ligado a cada juego y consiste en que al jugador se le desbloquean en su perfil ciertas hazañas después de conseguir superarlas. Algunos ejemplos de logros incluyen eventos individuales, como derrotar al

jefe final o acabar el juego en una dificultad concreta. Otros logros se dan como una estadística acumulativa en el tiempo, por ejemplo, ganar diez combates consecutivos. Esta funcionalidad intenta incentivar al jugador para alargar la vida útil de los juegos.

Los logros son una técnica para intentar que el jugador entienda mejor todo lo que le ofrece el juego, animar a los jugadores a explorar con más énfasis o probar estilos de juego completamente diferentes. Los logros permiten a los jugadores comparar su progreso entre sí y fomentar la competitividad entre ellos.

**Trofeos platino**

Una variante de los logros completados fueron los trofeos platino introducidos en PSN, que de una forma más visible muestran que el jugador ha completado todos los logros del juego.

Para entender cómo funcionan los logros, es necesario revisar los elementos básicos asociados a cada uno:

- **Identificador**: es una cadena única generada por la plataforma para referirse al logro entre sus clientes de juegos.
- **Nombre**: es un nombre corto del logro (por ejemplo, «Sin Rival»).
- **Puntos**: número de experiencia que se añadirá al perfil del jugador.
- **Descripción**: es una frase concisa acerca de su logro para ayudar al jugador a entender de qué se trata (por ejemplo, «acaba el juego en la dificultad más alta»).
- **Icono**: imagen asociada al logro.

La lista de logros de un juego muestra normalmente el icono, los puntos y el nombre de cada logro para los que han sido desbloqueados; en cambio, los logros bloqueados u ocultos aparecen con menor información.

Cada juego tiene una puntuación máxima (por ejemplo, 1.000 puntos, donde cada logro aporta una parte –entre 5 y 200 puntos, por ejemplo–).

Por otro lado, los logros pueden ser designados como estándar o como incremental. Generalmente, un logro incremental implica que un jugador progresará gradualmente hasta conseguir el logro durante un período de tiempo más largo. A medida que el jugador avanza hacia el logro incremental, el juego debe informar del progreso parcial del jugador a la plataforma de servicios. En cualquier caso, una vez informada la plataforma, esta se encarga de validar el logro y retornar la respuesta en caso afirmativo al juego, para que este pueda notificar al jugador en el momento que consigue el logro y pueda «saltar».

### 1.4. Clasificaciones (*leaderboards*)

Las **tablas de clasificación** o **marcadores** son una forma de proporcionar clasificaciones para ciertos aspectos de un juego, como por ejemplo, una puntuación o tiempo para completar un nivel en particular.

En general, los *rankings* o marcadores pueden ser consultados tanto en términos de un rango global como en rangos con filtros concretos, por ejemplo, en la relación con los amigos del jugador en la misma plataforma.

Por ejemplo, los marcadores en Steam pueden crearse a través del sitio web de gestión para desarrolladores de Steamworks o pueden crearse mediante programación a través de una llamada de su SDK. Esta forma de trabajar es muy común en el resto de plataformas.

El proceso típico funciona de esta manera: al final de un juego (o en un momento apropiado que hayas determinado), el juego envía la puntuación del jugador a una o más tablas de clasificación que hayas creado para el juego. Los servicios de juegos comprueban si este puntaje es mejor que la entrada de clasificación actual del jugador para la puntuación diaria, semanal o de todos los tiempos. Si es así, los servicios de juegos actualizan las tablas de clasificación correspondientes con la nueva puntuación.

Para recuperar los resultados de un jugador para una tabla de clasificación, se puede solicitar un marco de tiempo (diario, semanal o de todos los tiempos) y especificar si el usuario desea ver una tabla de clasificación social o pública. El servicio de juegos realiza todo el filtrado necesario y, a continuación, envía los resultados al cliente.

Un aspecto interesante de las tablas de clasificación es la posibilidad de asociar o subir contenido generado por el juego asociado con una entrada de tabla de clasificación. Por ejemplo, un *speedrun* de un nivel podría tener una captura de pantalla asociada o un vídeo mostrando cómo se ha conseguido.

***Speedrun***

Es una competición cuyo objetivo principal es acabar un videojuego lo más rápido posible, generalmente en dificultad máxima.

Alternativamente, un juego de carreras o aventuras podría proporcionar corredores fantasma de otros jugadores, que se pueden descargar y competir contra ellos. Esto permite formas de hacer que las tablas de clasificación sean más interactivas que simplemente listar las mejores puntuaciones.



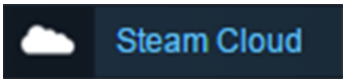
Marcadores y modo *Time Trial* con fantasma de *Mirror's Edge*

Las tablas de clasificación pueden ser otra forma de impulsar la competencia entre los jugadores, tanto para los aficionados más *hardcore*, que lucharán por el primer puesto en una clasificación pública, como para los jugadores más *casual*, que estarán interesados en comparar su progreso entre sus amigos.

## 1.5. Persistencia en la nube (*cloud saving*)

El servicio ofrece una manera simple de guardar la progresión de los jugadores de forma remota. En concreto, este servicio permite sincronizar los datos de juego de un jugador en varios dispositivos. Al integrar nuestro videojuego con este servicio, nos permite recuperar los datos guardados para los jugadores, que pueden continuar con el juego en su último punto de guardado.

Un ejemplo de persistencia en la nube es el servicio de *Steam Cloud*.




***Steam Cloud***

Por ejemplo, si un juego se ejecuta en una consola, se puede usar el servicio de guardado para permitir que el jugador inicie su juego en otra consola o incluso en un ordenador, siempre y cuando ambos permitan conectarse al mismo servicio (por ejemplo, Xbox Live) y luego continuar jugando sin perder ningún progreso.

Este servicio también se puede utilizar para asegurar que el juego de un jugador continúa desde donde lo dejó, incluso si su dispositivo se pierde, se rompe o lo renueva por uno más moderno.

El espacio que acostumbran a ocupar las partidas guardadas tiende a ser reducido, por lo tanto, este servicio no tiene un coste adicional por cantidad de datos guardados en la nube. Normalmente, este espacio ocupado se asocia a la cuota del jugador, que está dimensionada convenientemente.

Los juegos pueden seguir trabajando con su forma tradicional de leer y escribir las partidas en local, especialmente cuando el dispositivo del jugador está sin conexión. La idea es que el juego, al detectar la conexión con la plataforma, ejecute un proceso de sincronización para que de forma asíncrona los datos del juego sean consistentes en los servidores remotos.

Obviamente, al utilizar este servicio, el juego puede encontrar conflictos a la hora de guardar datos, ya que el jugador puede haber jugado en un dispositivo *offline* a la vez que lo hacía en otro dispositivo en línea. En general, la mejor manera de evitar conflictos de datos es cargar siempre las partidas más recientes del servicio cuando la aplicación se inicia o se reanuda y se guardan los datos en el servicio con una frecuencia razonable. Sin embargo, no siempre es posible evitar conflictos de datos.

El juego debe hacer todo lo posible para manejar los conflictos de manera que los datos de sus usuarios se conserven y que tengan una buena experiencia. Esta situación también nos la podemos encontrar con otros servicios, como marcadores o logros, de forma que se pueda concentrar la sincronización de datos en el mismo proceso, aunque es mucho más crítico en el caso de las partidas al tratarse de información más sensible y específica.

### 1.6. *Matchmaking*

Este mecanismo está muy ligado a los juegos multijugador, donde se requiere emparejar o agrupar a los jugadores para realizar partidas equilibradas o no tanto, públicas o privadas, etc. El flujo básico de preparación para jugar un juego multijugador es aproximadamente como sigue:

- El juego busca un *lobby* o vestíbulo basado en parámetros personalizables del juego. Estos parámetros pueden incluir modos de juego o incluso nivel de habilidad (si se realiza *matchmaking* basado en habilidades).

- Si se encuentran uno o más vestíbulos adecuados, el juego selecciona uno automáticamente o el jugador puede seleccionar uno de una lista. Si no se encuentra ningún vestíbulo, el juego puede elegir crear uno para el jugador. En cualquier caso, cuando se ha encontrado o creado el vestíbulo, el jugador puede unirse a él.

- En el vestíbulo es posible configurar aún más los parámetros del próximo juego, como personajes, mapa, etc. Durante este período, otros jugadores esperanzadamente se unirán al mismo vestíbulo. También es posible enviarse mensajes de chat entre sí, mientras los jugadores están en el mismo *lobby*.

- Una vez que el juego está listo para comenzar, los jugadores se unen a su juego y salen del vestíbulo. Normalmente, esto implica la conexión a un servidor de juego (ya sea un servidor dedicado o un servidor de juego). En el caso de no tener servidor, los jugadores empiezan a comunicarse entre sí *peer-to-peer* antes de salir del vestíbulo.

A continuación, vamos a estudiar un sistema concreto de *matchmaking* para entender en qué consiste, qué particularidades o técnicas se han definido y cómo se aplica.

**Equilibrio en *matchmaking***

Predecir la probabilidad de cada resultado del juego mejora la competitividad *matchmaking*, lo que permite reunir equipos con habilidades más equilibradas.

Tomando como base el sistema de puntuación Glicko, un método para evaluar la habilidad de un jugador en juegos de habilidad, el conocido sistema TrueSkill presenta un algoritmo bayesiano de clasificación y un sistema de *matchmaking* desarrollado por Microsoft Research y establecido en los servicios en vivo de Xbox 360. Con TrueSkill, un jugador recién llegado a la plataforma puede clasificarse correctamente en menos de veinte juegos. En TrueSkill el rango de un jugador se representa como una distribución normal $\aleph$ caracterizada por un valor medio de $\mu$ (habilidad percibida) y una varianza de $\sigma$ (la seguridad depositada en el valor $\mu$).

**TrueSkill**

Utiliza un modelo matemático de incertidumbre para abordar debilidades en sistemas de clasificación existentes.

Cuando se produce una coincidencia, el sistema intenta agrupar individuos según su nivel de habilidad estimado. Si dos individuos compiten cara a cara y tienen el mismo nivel de habilidad estimado con una incertidumbre de estimación baja, cada uno tiene aproximadamente una probabilidad del 50 % de ganar una partida. De esta manera, el sistema intenta hacer cada partida tan competitiva como sea posible.

Con el fin de evitar el abuso del sistema, la mayoría de los juegos clasificados tienen opciones relativamente limitadas para *matchmaking*. Por diseño, los jugadores no pueden jugar fácilmente con sus amigos en los juegos clasificados. Sin embargo, estas contramedidas han fracasado debido a técnicas como cuentas alternativas y fallos del sistema en los que cada sistema tiene su propia calificación TrueSkill individual. Para proporcionar juegos menos competitivos, el sistema permite también juegos con partidas sin clasificar, donde se emparejan jugadores de cualquier nivel de habilidad, sin que estos contribuyan a la calificación de TrueSkill.

## 1.7. Otros servicios

A parte de todos estos servicios, podemos encontrar algunos similares o más bien específicos de red, que suelen diferir más dependiendo de la plataforma escogida. En este último apartado vamos a ver brevemente en qué consisten los servicios que proporcionan un envoltorio para la comunicación en red entre los jugadores. En el caso de Steamworks o Google Play Games, se proporciona una serie de funciones para enviar paquetes a otros jugadores creando una red *peer-to-peer*.

Concretamente, esta vez pondremos como ejemplo la API multijugador en tiempo real de Google Play Game. Esta es lo suficientemente flexible como para que podamos utilizarla para reescribir nuestro juego y hacer que todas las instancias del juego envíen los datos a través de la red *peer-to-peer* subyacente.

**Elección de líder**

Para escoger al *host* se puede hacer uso de un algoritmo de elección de líder.

Un tema importante a tener en cuenta es que una vez designado el cliente que actuará como *host* y se establezca su autoridad en los datos del juego, este será el encargado de centralizar la comunicación y transmitir datos a los demás participantes conectados a través del sistema de mensajería de datos de Google Play Games.

Además, una vez superada la fase de *matchmaking* para crear la partida, si la lógica de juego se basa en la existencia de un «anfitrión» o «propietario» del juego, será el juego y no la plataforma el responsable de implementar el algoritmo para determinar quién es el *host*.

Como hemos dicho, si reescribimos nuestro juego para ello, podemos utilizar los servicios de juegos de Google Play para transmitir datos a los participantes en una partida o permitir a los participantes intercambiar mensajes entre sí. Como en otros servicios, los mensajes de datos pueden enviarse utilizando un protocolo de mensajería fiable o poco fiable proporcionado por los servicios de la plataforma:

- **Mensajería fiable**: la entrega de datos, la integridad y el orden están garantizados. Podemos elegir notificar el estado de entrega mediante una devolución de llamada. La mensajería fiable es adecuada para enviar datos no sensibles al tiempo. También se puede usar mensajería fiable para enviar grandes conjuntos de datos, donde estos se pueden dividir en segmentos más pequeños, son enviados a través de la red y luego reensamblados por el cliente receptor. Este tipo de mensajería puede tener latencia alta.

- **Mensajes poco fiables**: el cliente del juego envía los datos una sola vez (*fire-and-forget*) sin garantía de entrega de datos o datos que lleguen en orden. Sin embargo, la integridad está garantizada, por lo que no es necesario agregar un código de verificación o *checksum*. Ese tipo de mensajería tiene una latencia baja y es adecuada para enviar datos sensibles al tiempo. Su aplicación es responsable de garantizar que el juego se comporte correcta-

mente si los mensajes se descartan en transmisión o se reciben fuera de servicio.

Obviamente, si un cliente que envía datos no está conectado a los servicios de juego de Google Play o el destinatario no está conectado, el mensaje no se entregará. Para conservar las transmisiones de mensajes y evitar exceder los límites de velocidad, las prácticas recomendadas para enviar datos son las siguientes:

- Enviar mensajes solo a los participantes que requieran esa información, en lugar de transmitir a todos los participantes. Si se está enviando un mensaje de difusión, hay que excluir al participante del remitente de la lista de destinatarios de difusión.

- Si se están enviando datos utilizando el protocolo de mensajería fiable, hay que mantener la frecuencia de las transmisiones de mensajes por debajo de un límite razonable (por ejemplo, cincuenta mensajes por segundo). Si se necesita enviar datos con más frecuencia, es recomendable utilizar mensajería no fiable.

# 2. *Cloud computing*

Desde hace unos años, y gracias a la computación en la nube, se ha facilitado mucho a los desarrolladores poder hospedar, gestionar y mantener servicios remotos. Incluso los pequeños estudios pueden permitirse sus propios servidores dedicados. En este apartado veremos en qué consiste este paradigma, y de qué forma podemos aprovecharnos de él para que nuestro juego, parte de este o servicios asociados se ejecuten en servidores remotos.

## 2.1. Descripción conceptual

Las plataformas en la nube han tomado mucha relevancia estos años gracias a sus ventajas inherentes, de manera que han transformado los servicios de *hosting* tradicionales en otra forma de venderse al gran público, en parte gracias a proporcionar un servicio con bajos costes de gestión y administración de las infraestructuras de hardware.

Otra razón importante para adoptar los sistemas en nube es su capacidad para escalar rápidamente en entornos de rápido crecimiento con cargas de trabajo masivas. Alcanzar la escalabilidad deseada de forma flexible es una tarea compleja, que implica resolver muchos problemas concurrentes, como mantener el equilibrio de carga de los recursos, políticas de replicación y coherencia y escalabilidad horizontal de almacenes de datos persistentes.

En estos casos, los proveedores de servicios en la nube deben ofrecer *frameworks* con API específicas para dichos servicios, como por ejemplo Microsoft Azure, Amazon EC2 o Google App Engine. Las tecnologías de la nube comenzaron a emerger cuando Amazon irrumpió estelarmente en el mercado en 2006 ofreciendo sus servicios web. Y de hecho, Amazon se ha convertido en un estándar de nube de facto, aunque ahora muchas otras compañías proporcionan soluciones en un ecosistema empresarial muy activo.

Pero existen diferentes tipos de de nube y formas de diferenciarlos según su naturaleza.

De acuerdo con la definición ampliamente utilizada por el NIST (National Institute of Standards and Technology):

> «El *cloud computing* es un modelo para habilitar el acceso a la red de forma omnipresente, conveniente y bajo demanda a un conjunto compartido de recursos computacionales configurables (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que se pueden aprovisionar y liberar rápidamente con un mínimo esfuerzo de gestión o interacción con el proveedor del servicio».

Dependiendo de la capacidad proporcionada al consumidor podemos distinguir entre tres modelos de servicio:

- **Software como servicio (SaaS)**: utilizar las aplicaciones del proveedor que se ejecutan en una infraestructura en la nube. Las aplicaciones son accesibles desde varios dispositivos cliente a través de una interfaz de cliente ligero, como un navegador web (por ejemplo, correo electrónico basado en web) o una interfaz de programa. El consumidor no gestiona ni controla la infraestructura subyacente de la nube, incluyendo la red, los servidores, los sistemas operativos, el almacenamiento o incluso las capacidades de las aplicaciones individuales, con la posible excepción de los ajustes de configuración específicos de la aplicación del usuario.

- **Plataforma como servicio (PaaS)**: desplegar en la infraestructura de nube las aplicaciones creadas o adquiridas por el consumidor utilizando lenguajes de programación, bibliotecas, servicios y herramientas soportadas por el proveedor. El consumidor no gestiona ni controla la infraestructura subyacente de la nube, servidores, sistemas operativos o almacenamiento, pero tiene control sobre las aplicaciones desplegadas y posiblemente configuraciones para el entorno de hospedaje de aplicaciones.

- **Infraestructura como servicio (IaaS)**: proveer procesamiento, almacenamiento, redes y otros recursos de computación fundamentales donde el consumidor es capaz de desplegar y ejecutar software arbitrario, que puede incluir sistemas operativos y aplicaciones. El consumidor no gestiona ni controla la infraestructura subyacente de la nube, sino que tiene control sobre los sistemas operativos, el almacenamiento y las aplicaciones implementadas, y posiblemente un control limitado de los componentes de red selectos (por ejemplo, *firewalls* de *host*).

Según el uso de la infraestructura de la nube, podemos clasificar los modelos de implementación en los cuatro siguientes:

- **Nube privada**: uso exclusivo por una única organización que comprende múltiples consumidores (por ejemplo, unidades de negocio). Puede ser propiedad, administrada y operada por la organización, un tercero o una combinación de ellos, y puede existir dentro o fuera de las instalaciones.

- **Nube pública**: uso abierto por el público en general. Puede ser propiedad, administrada y operada por una organización comercial, académica o gubernamental, o alguna combinación de ellas. Se sitúan físicamente en las instalaciones del proveedor de la nube.

- **Nube híbrida**: es una composición de dos o más infraestructuras de nube distintas que siguen siendo entidades únicas, pero están unidas por una

tecnología estandarizada o propietaria que permite la portabilidad de datos y aplicaciones.

## 2.2. Características de los sistemas *cloud*

**El caso *Pokémon GO***

Aún a fecha de hoy es complicado adivinar el número de usuarios. En 2016, Niantic subestimó en una escala de magnitud (x10) los jugadores de *Pokémon GO*.

En los primeros días de los juegos en línea, el alojamiento de los propios servidores dedicados era tarea muy compleja, empezando por la adquisición y el mantenimiento de grandes cantidades de hardware, infraestructura de redes y personal de TI. La estimación de jugadores en el lanzamiento era sumamente importante, ya que una infraestructura sobredimensionada podía significar un coste adicional inmantenible. Pero peor aún era quedarse corto y que los jugadores pagaran sin poder conectarse debido a restricciones de procesamiento y ancho de banda.

Pero como hemos visto, la nube ha solventado esos problemas. Gracias al abundante poder de procesamiento a la carta disponible en los gigantes de Internet, como Amazon, Microsoft y Google, las compañías de juegos son capaces de manejar estos costes de forma mucho más razonable. Otros servicios de terceros, como Heroku, facilitan el despliegue proporcionando servicios de administración de servidores y bases de datos según sea necesario. Además, existen servicios orientados directamente a videojuegos, como es el caso de Unity Cloud o Photon Cloud, que nos permiten hospedar los juegos con servicios más orientados y dedicados a las necesidades concretas de los videojuegos.

A pesar de la falta de coste inicial en el lado del servidor, seguimos teniendo algunos inconvenientes potenciales que debemos considerar:

- **Complejidad**: utilizar un grupo de servidores dedicados es una tarea compleja, a pesar de que la nube proporciona la infraestructura y parte del software de administración. Pero además se requiere código personalizado de administración de procesos y máquinas virtuales, así como adaptarse a las API cambiantes.

- **Coste**: aunque la nube disminuye el coste inicial y de largo plazo de manera significativa, no es totalmente gratuito. El aumento del interés del jugador puede cubrir el aumento del coste, pero depende de cada caso.

- **Dependencia de terceros**: hospedar el juego en los servidores de Amazon, Microsoft o Unity significa que el destino descansa en sus manos. Aunque las empresas ofrecen acuerdos de nivel de servicio que garantizan un mínimo de tiempo de actividad, sigue habiendo un riesgo asociado.

- **Cambios inesperados de hardware**: los proveedores de *hosting* generalmente garantizan proporcionar hardware que cumple con ciertas especificaciones mínimas. Esto no les impide cambiar de hardware sin previo aviso, siempre y cuando esté por encima de la especificación mínima. Si

de repente introducen una configuración de hardware extraña que no han probado, puede causar problemas.

Aunque estas desventajas pueden ser significativas, los beneficios a menudo las superan:

- **Servidores fiables, escalables y de alto ancho de banda**: en modelos tradicionales no hay garantías de que los jugadores anfitriones sean los que tengan mejor ancho de banda cuando sus otros jugadores quieren jugar. Con alojamiento en la nube y un buen programa de administración de servidores, se puede activar cualquier servidor donde y cuando se necesite para realizar este tipo de tareas y asegurar el buen funcionamiento de la partida.

- **Prevención de trampas**: al controlar todos los servidores es fácil asegurarse de que están ejecutando versiones no modificadas y legítimas de los juegos. Esto significa que todos los jugadores obtienen una experiencia uniforme y fiable. Esto permite no solo clasificaciones reales, sino todos los servicios de la plataforma que utilicen los juegos.

- **Protección de copia razonable**: restringir los juegos a ser ejecutados en servidores dedicados proporciona una forma de DRM *de facto*, no intrusiva. Sin liberar ejecutables de servidor dificulta el despliegue de servidores pirata o el desbloqueo de contenido de forma ilegal. También permite comprobar las credenciales de inicio de sesión para cada jugador, asegurándose de que realmente deberían estar jugando su juego.

En conclusión, la opción de hospedar servidores dedicados puede estar por encima de su nivel de pago. Sin embargo, teniendo en cuenta el valor de los ingenieros de pila completa en la fuerza de trabajo, es importante entender todas las implicaciones de la decisión para que se pueda sopesar con una opinión informada sobre la base de los detalles del juego que nuestro equipo está haciendo.

### 2.3. Integración con sistemas *cloud*

Cuando se trabaja en un nuevo entorno, es más eficiente trabajar con herramientas adaptadas a ese entorno. El desarrollo de servidores *backend* es un campo en evolución constante y veloz, que consta de un conjunto de herramientas que evoluciona rápidamente.

Por suerte, existen muchos lenguajes, plataformas y protocolos diseñados para hacer la vida más fácil para el desarrollador de *backend*. Los *frameworks* modernos de desarrollo web pueden ser vistos como un conjunto de componentes que simplifican la programación HTTP, controlando automáticamente los detalles de transporte de bajo nivel.

Además, en los últimos años se ha puesto de manifiesto que HTTP no debe usarse solo para servir páginas HTML. También es un gran protocolo para la creación de una API estándar vía REST, utilizando los verbos (GET, POST, etc.) y códigos (200, 500, etc.) proporcionados, además de algunos conceptos simples, como URI y encabezados.

El estilo arquitectónico REST ha demostrado ser una forma eficaz de aprovechar el HTTP, aunque ciertamente no es el único enfoque válido para HTTP. Actualmente, existe una tendencia clara para que los servicios utilicen API REST y datos JSON, junto con *WebSockets* para complementar los servicios más interactivos.

Estas son herramientas flexibles y ampliamente aceptadas para el desarrollo de servidores. Es viable elegir diferentes herramientas para nuestra plataforma *cloud* de servicios, pero claramente para poder realizar la integración de nuestros videojuegos se requiere una comprensión básica de HTTP, JSON, REST y *WebSockets*, como las principales tecnologías web involucradas.

### 2.3.1. El estándar JSON

A finales de los años noventa y principios de los dos mil, el lenguaje XML (*eXtensible Markup Language*) fue anunciado como el formato universal de intercambio de datos que cambiaría la forma de estructurar documentos. Los inicios fueron buenos, y muchas tecnologías se apuntaron a usar la que fue durante ese periodo la única opción para compartir datos libremente.

En la actualidad, JSON (*JavaScript Object Notation*) es el formato ligero que se ha convertido en el favorito para el intercambio de datos universal, soportando todos los tipos de datos básicos, como booleanos, texto, números, matrices y objetos.

> JSON es la herramienta por defecto para compartir datos. Aunque XML tiene sus ventajas, es más fácil trabajar con datos almacenados en una estructura más integrada con los lenguajes orientados a objetos. Esto hace que sea muy sencillo importar y exportar datos desde un documento JSON, ya que para hacer lo mismo con XML se precisa una transformación complementaria.

JSON se basa en texto, manteniendo toda la legibilidad humana de XML, pero con menos requisitos de formateo y cierre de etiquetas. Esto hace aún más agradable leer y depurar. Funciona bien como formato de datos en los documentos asociados a peticiones web (encabezado HTTP especificando el *ContentType* de aplicación JSON), tanto de escritura (creación POST, actualización PUT o actualización parcial PATCH) como de consulta (GET).

La utilidad principal desde el punto de vista de los videojuegos multijugador es la de facilitar las tareas de serialización y compresión de objetos a documentos textuales y viceversa, entre clientes y servidores. A continuación vamos a ver cómo trabajar con este formato en C#, a partir de la noción de JSON «estructurado», la cual significa que se debe indicar qué variables se van a almacenar en sus datos JSON creando una clase o estructura. Por ejemplo:

```
1.      [Serializable]
2.      public class MyClass {
3.          public int level;
4.          public float timeElapsed;
5.          public string playerName;
6.      }
```

Definimos una clase C# simple que contiene una serie de variables (por ejemplo, *level*, *timeE-lapsed* y *playerName*) y la marcamos como *serializable*, lo cual es necesario para poder usar el serializador JSON. A continuación, vamos a crear una instancia de esta clase y serializarla en formato JSON mediante la función *JsonUtility.ToJson()*:

```
7.      MyClass myObject = new MyClass();
8.      myObject.level = 1;
9.      myObject.timeElapsed = 47.5f;
10.     myObject.playerName = "Dr Charles Francis";
11.
12.     string json = JsonUtility.ToJson(myObject);
```

Esto resultaría en la variable JSON, que contiene la cadena de caracteres:

```
13.     {"level":1,"timeElapsed":47.5,"playerName":"Dr Charles Francis"}
```

Finalmente, para convertir el JSON de nuevo en un objeto podemos utilizar la función *JsonUtility.FromJson()*:

```
14.     myObject = JsonUtility.FromJson<MyClass>(json);
```

De esta forma podemos comprender la sencillez de trabajar con esta herramienta: es mucho más compleja la tarea de enviar y recibir datos vía HTTP, REST o *WebSockets*, que la de codificarlos en formato JSON.

### 2.3.2. Conexiones *WebSockets*

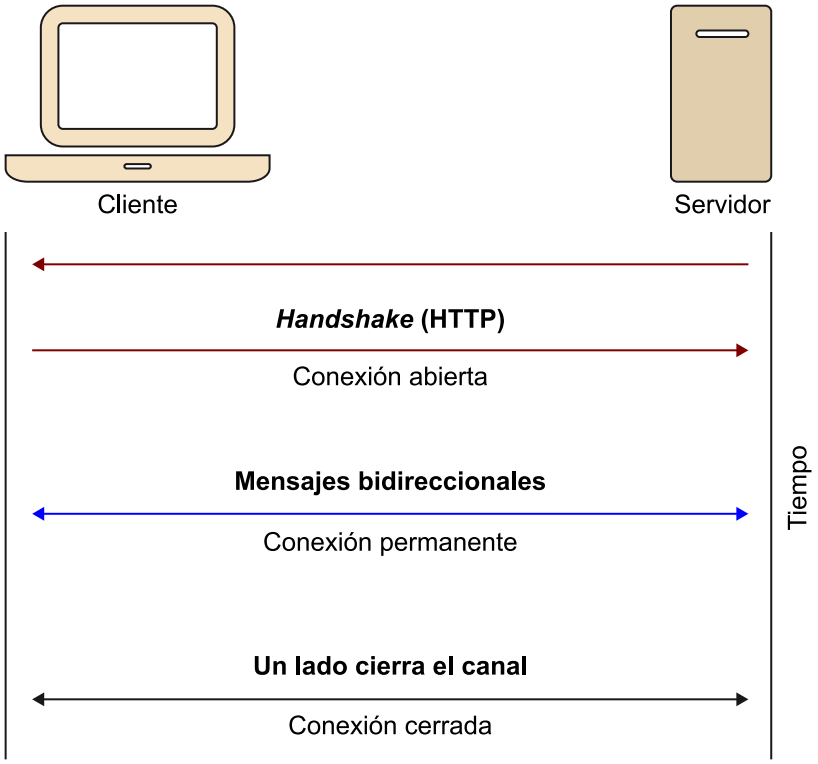
*WebSockets* es una extensión de la idea de *socket* tradicional, motivada por las carencias inherentes del protocolo HTTP, el cual fue diseñado para la *World Wide Web* y, por lo tanto, para ser usado por los navegadores.

Al ser un protocolo específico que funciona de una forma particular, no resulta adecuado para cada necesidad, concretamente nos referimos a la forma en la que HTTP manejaba las conexiones, donde se realiza una conexión por cada solicitud, por ejemplo, para descargar un documento HTML o una imagen. Se abre siempre un puerto/*socket* y se transfieren los datos, cerrando la conexión justo al acabar. Esta continua apertura y cierre de conexiones de HTTP puede crear una sobrecarga, y para ciertas aplicaciones, especialmente aquellas que requieren respuestas rápidas o interacciones en tiempo real o mostrar flujos de datos, simplemente no funciona.

La otra limitación con HTTP es que pertenece al **paradigma *pull***, donde es el interesado, en este caso el navegador, quien solicita o extrae información del origen, en este caso los servidores. El problema aquí es que el servidor solo es capaz de enviar datos al navegador en el momento de responder a una solicitud. Esto significa que los navegadores deben consultar al servidor de forma continua para obtener nueva información, repitiendo las solicitudes cada tantos segundos o minutos para averiguar si hay nuevos datos.

A finales de los años dos mil surgió un movimiento para añadir *sockets* a los navegadores, y en 2011 el protocolo *WebSocket* fue estandarizado. Esto permitió a los desarrolladores usar dicho protocolo, que gracias a su flexibilidad permite crear nuevas formas de comunicación para transferir datos desde y hacia servidores desde el navegador, así como comunicación *peer-to-peer* (P2P) o comunicación directa entre navegadores. La principal diferencia con el protocolo HTTP es que un *WebSocket* permanece abierto al servidor teniendo siempre un canal disponible para la comunicación bidireccional. Esto significa que los datos pueden ser enviados al navegador en tiempo real bajo demanda, cambiando así la forma de trabajar del paradigma *pull* al **paradigma *push***.

Diagrama de conexión WebSocket




### 2.3.3. La arquitectura REST

El REST, o *REpresentational State Transfer*, es otra abstracción para crear API para aplicaciones de una manera estandarizada.

En las aplicaciones web típicas y tradicionales, la creación de *endpoints* REST mediante HTTP es la forma en que se diseñan la gran mayoría de las aplicaciones. Cualquiera de las diversas tecnologías disponibles (por ejemplo, JavaScript) es muy similar en la forma en que se reciben las solicitudes de información y luego se responde a estas solicitudes.

REST organiza estas peticiones de forma predecible, usando tipos de operaciones HTTP, o verbos, para construir respuestas apropiadas. Las peticiones se originan en el cliente y los verbos HTTP más comunes son GET, POST, PUT, DELETE, aunque hay muchos otros. Corresponden a operaciones esperadas, recuperación de datos, envío de datos, actualización de datos y eliminación de datos.

REST es, de lejos, la forma más estandarizada de estructurar la API para peticiones web. Fácilmente podemos encontrar ejemplos en muchas aplicaciones web que hacen uso de este sistema y exponen su API para poder realizar peticiones. Un ejemplo típico sería Twitter, que además nos permite ligarlo con *OAuth* para poder utilizar su sistema de usuarios desde nuestra aplicación y acceder posteriormente a sus datos. Esto sería útil para poder tuitear los resultados de nuestras partidas de forma automática desde el juego.

***OAuth***

Es un sistema que se combina perfectamente con REST para permitir la autentificación usando sistemas de terceros (por ejemplo, Google o Facebook) desde una aplicación.

Pero dado que implica el uso de HTTP, también tiene la sobrecarga asociada a ese protocolo. Para la mayoría de las aplicaciones, la información solo necesita ser transferida cuando un usuario realiza una acción. Por ejemplo, al navegar por un sitio de noticias, una vez que el navegador ha solicitado el artículo, el usuario está ocupado leyéndolo y no realiza otras acciones. Tener el puerto/*socket* cerca durante este tiempo es realmente ahorrar recursos. Con una interacción menos frecuente, HTTP funciona muy bien, y es por eso que se utiliza.

Para más interacción en tiempo real, o transferencia en tiempo real de datos, HTTP y REST no son la combinación de protocolo y abstracción más adecuada. Aquí es donde los *sockets*, y en este caso concreto *WebSockets*, son mucho más adecuados.

# 3. Proyecto: *Tanks!* en línea

En este último proyecto vamos a integrar el juego *Tanks!* con un servicio de chat remoto y escalable. El principal objetivo de este ejemplo es ver cómo utilizar herramientas actuales para poder añadir funcionalidades remotas a nuestros desarrollos, como comunidades, soporte a retransmisiones en directo, mundos persistentes, algoritmos de física o inteligencia artificial distribuidos, etc. Así pues, el propósito del proyecto no es tanto añadir la funcionalidad de chat en sí, sino que sirve de justificación para ver cómo integrar nuestro juego en un servicio en línea. La elección de una herramienta de chat, entre otras, viene dada simplemente por su sencillez.

## 3.1. Lado servidor

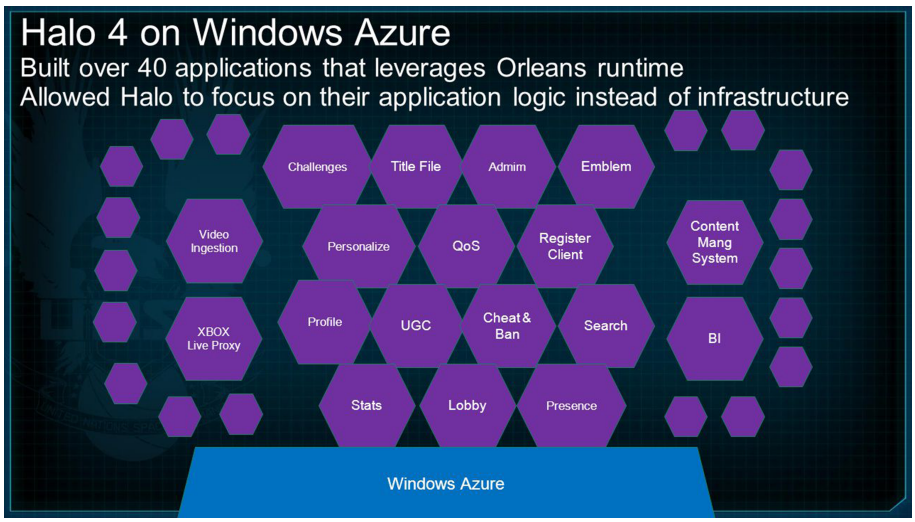
Como hemos visto, actualmente existen diferentes opciones para crear nuestro servidor o servidores, siendo la computación en la nube la forma más flexible y escalable a la que podemos optar. Esta suele estar asociada con coste mensual, como también lo tenemos en el *hosting* tradicional, pero más orientada al consumo que nuestra plataforma realice.

Es por ello que para este proyecto hemos optado por una solución de tipo PaaS (plataforma para desplegar nuestro código), pero con la modalidad de nube privada, donde la lógica que requieran nuestros videojuegos se ejecutará de forma remota pero en nuestros servidores y será administrada por nosotros.

**PaaS**

Una plataforma en la nube donde podemos desplegar nuestro código siempre que este esté soportado por el proveedor.

Para este fin vamos a hacer uso de una plataforma real de distribución libre basada en el proyecto Orleans de Microsoft. Este proyecto fue originalmente concebido para dar soporte a las características en línea de *Halo 4* a partir de 2012, dentro de la nube propietaria de Microsoft (Azure), y ha ido evolucionando hacia el proyecto independiente y de código libre que resulta hoy en día.




Arquitectura del proyecto Orleans

Por su complejidad, no usaremos Orleans, sino que trabajaremos con su versión en Java llamada Orbit, que está básicamente basado en Orleans, pero fue desarrollado por Bioware originalmente para dar soporte a *Dragon Age Inquisition*. Posteriormente ha pasado a ser utilizado por Electronic Arts para otros de sus juegos.

**Orbit**

Para trabajar con Orbit, el primer paso es instalar el entorno de desarrollo de Java SDK 8: https://www.java.com.

### 3.1.1. Problemática

Esta arquitectura difiere de la clásica de tres capas, que se ya ha demostrado que no funciona para todos los escenarios. Concretamente, en el caso que nos ocupa, precisamente debemos estar preparados para el peor de los escenarios y no morir de éxito. Un claro ejemplo donde no se calculó bien la previsión y que ya hemos comentado fue *Pokémon GO*.

**Proyecto Orbit**

Podéis acceder al repositorio oficial de código abierto del proyecto Orbit en GitHub: https://github.com/orbit/orbit.

En realidad, a pesar de que se tomaron medidas para lanzar el juego de forma escalonada por países, en su primera semana en la calle alcanzó una carga de 45X. Esto ocasionó muchos problemas y fallos de conexión en los *early-adopters*[1] del juego, dejando unas críticas negativas en las diferentes tiendas de las plataformas móviles donde fue lanzado. Obviamente, los recursos de hardware detrás de este videojuego, con Google, fueron de gran presupuesto.

(1)Término usado para referirse a los clientes que se lanzan a por el producto antes que nadie.

Se trata pues de un caso extremo, pero no hay que subestimar que podamos llegar a desarrollar y lanzar un juego donde este pico inicial de conectividad de los primeros días sea uno de los problemas a solucionar más importantes. Evitar las críticas de los *early-adopter* es muy importante siempre, ya que dejan una mancha difícil de borrar para el resto de la vida útil de un videojuego.

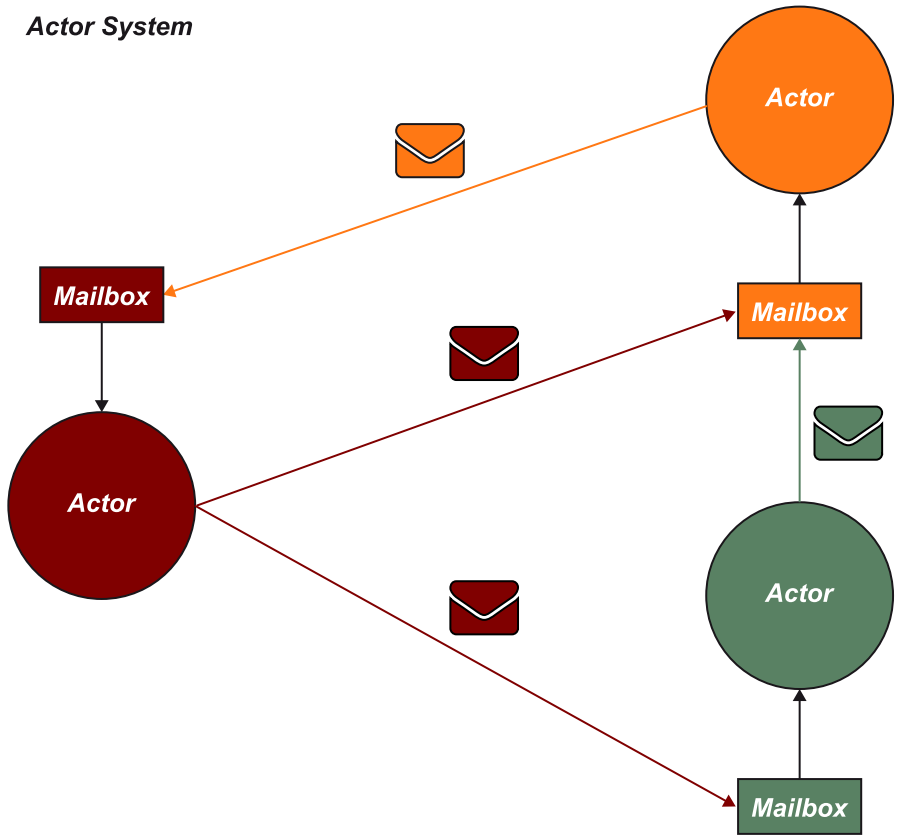
Esto no significa que en casos más modestos no debamos estar preparados para absorber una cantidad elevada de jugadores con el mínimo número de problemas posibles.

### 3.1.2. Arquitectura

Orbit nos permite construir nuestro propio sistema en la nube para dar servicio a nuestros juegos. Esta solución nos permite construir un juego con acceso a servicios en línea (por ejemplo, clasificaciones), partes del juego en la nube (por ejemplo, IA) o directamente juegos persistentes donde la mayor parte de la lógica esté fuera del juego.

La idea es utilizar el *actor model* para eliminar problemas inherentes a los sistemas distribuidos, gracias al hecho de centrarnos en la separación de responsabilidades, es decir, qué funcionalidad es la que queremos ofrecer al cliente. Estas están perfectamente aisladas, pero con la idea de que la lógica y los datos de cada servicio se encuentren en la misma entidad y que todo el sistema trabaje de forma asíncrona y en procesos individuales. Además, cada proceso es distribuido, de forma que un actor se encuentra en una máquina, pero evitando el concepto de servidor tradicional, no nos preocupamos de este, sino del acto que ofrece el servicio que los clientes solicitan, activándose automáticamente en la máquina o máquinas que sea necesario.

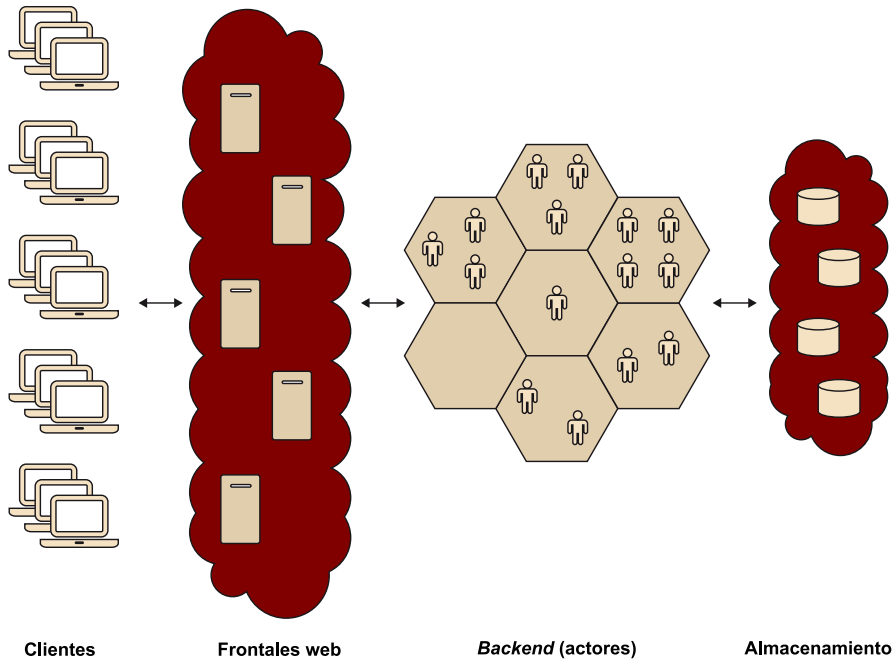
Idea principal del *actor model*



En este modelo de actores podemos ver bastantes similitudes con los *GameObjects* en remoto y las llamadas RPC entre ellos. La principal diferencia es que la comunicación es siempre asíncrona, basándose en un sistema de mensajes. Por lo tanto, seguimos trabajando a nivel de aplicación, sin tener en cuenta detalles de las capas inferiores, como qué protocolo se utiliza para el transporte o la gestión de las IP de cada máquina, de forma que será tan transparente que no sabremos (ni necesitamos saber) en qué máquina se está ejecutando cada actor en cada momento.

Otra característica importante que debemos añadir es la facilidad para ampliar la plataforma mediante extensiones. Actualmente hay disponibles diversas extensiones oficiales, que incluyen importantes funcionalidades, como herramientas de *stream*, serializador JSON, la posibilidad de añadir una base de datos NoSQL (por ejemplo, MongoDB) para invertir en escalabilidad también la capa de almacenamiento, o crear un clúster de servidores, todo ello de forma transparente para nuestro código de actores.

Diagrama de la arquitectura Orbit




**Enlace recomendado**

Encontraréis la documentación oficial de Orbit en su página wiki: https://github.com/orbit/orbit/wiki.

La formación de clústers en esta plataforma es muy sencilla gracias a la extensión proporcionada, que se basa en unificar servidores mediante el mismo identificador. Esta infraestructura nos permite construir nuestra propia plataforma en la nube, y es una solución escalable debido a que puede crecer de forma proporcional a la demanda que tenga.

**Enlace recomendado**

Este ejemplo de chat para Orbit, junto con sus *scripts* de arranque, lo podéis descargar de su repositorio de GitHub: https://github.com/orbit/orbit-samples/.

### 3.1.3. Funcionamiento

**Akka**

Es un *framework* de actores en Java (Scala) más longevo que Orbit que también ha sido utilizado para dar soporte a videojuegos.

Nuestra principal tarea es desarrollar la lógica a nivel de aplicación, gracias a la abstracción que nos proporciona Orbit a nivel de actores. A diferencia de otras soluciones similares (por ejemplo, Akka), esta solución también nos evita instanciar actores en nuestra plataforma, ya que proporciona un concepto de actores virtuales que ni se crean ni se destruyen, sino que su existencia es perpetua, su instanciación es totalmente automática y su localización transparente para nosotros.

Para definir un actor debemos extender la interfaz *Actor* y definir los métodos que este proporcionará para que los clientes puedan realizar las llamadas remotas hacia estos:

```
1.  public interface Chat extends Actor {
2.      @OneWay
3.      Task<Void> say(String playerId, String message);
4.      Task<Boolean> join(ChatObserver observer);
5.  }
```

**Enlace recomendado**

Para compilar y descargar las dependencias de este ejemplo, debéis instalaros primero la librería Maven: https://maven.apache.org/.

Como podemos ver, nuestros métodos deben retornar un objeto *Task*, que es el encargado de proporcionar el resultado una vez se haya completado la tarea, debido a la naturaleza asíncrona del sistema.

Otro dato importante es el parámetro que recibe en nuestro caso el método *join()*, en el que los clientes se suscriben a los mensajes del chat como un actor de tipo *observer*, que es la forma proporcionada para que el actor puede realizar las llamadas remotas de respuesta a los clientes (situados en el *front-end*):

```
6.  public interface ChatObserver extends ActorObserver {
7.  @OneWay
8.  Task<Void> receiveMessage(String message);
9.  }
```

Empezando con la implementación de estos métodos del *ChatActor*, primero debemos recalcar que en este caso se trata de un actor con estado persistente (*stateful*), lo que le permite guardar, entre otras cosas, los clientes registrados (mediante *observers*) y el histórico de los mensajes anteriores:

```
10.  public class ChatActor extends
11.      AbstractActor<ChatActor.State> implements Chat {
12.
13.      public static class State {
14.          ObserverManager<ChatObserver> observers =
15.              new ObserverManager<>();
16.          LinkedList<String> history = new LinkedList<>();
17.      }
```

Sobre los dos métodos comentados anteriormente, empezamos con *join*, donde se realiza simplemente la persistencia del *observer* de la siguiente forma: marcada por el *framework*, añadiendo un nuevo *observer* a la lista y retornando la *Task* un resultado siempre cierto una vez finalizada la operación por parte del actor responsable del chat:

**Sintaxis lambda**

En estos ejemplos de hace uso de la sintaxis lambda de Java, definiendo funciones anónimas como la mayoría los de lenguajes modernos.

```
18.     public Task<Boolean> join(final ChatObserver observer) {
19.         state().observers.addObserver(observer);
20.         return writeState().thenApply(x -> true);
21.     }
```

De forma similar, con el método *say()*, podemos ver que para enviar los mensajes se hace uso de la lista de *observers*, ejecutando el método definido para ello (*receiveMessage*) y finalizar la acción añadiendo el mensaje al histórico y persistir en el estado:

```
22.     public Task<Void> say(final String playerId,
23.         final String message) {
24.
25.         state().observers.notifyObservers(o ->
26.             o.receiveMessage(message));
27.         state().history.add(playerId + "> " + message);
28.          return writeState();
29.     }
```

Finalmente, para hacer uso de este actor desde la plataforma, debemos implementar nuestro propio *observer* donde se encuentre la lógica del cliente (por ejemplo, *ChatMesageObserver*) y, situándonos en el frontal, necesitamos obtener la referencia remota del *ChatActor* para llamarlo y suscribir al cliente:

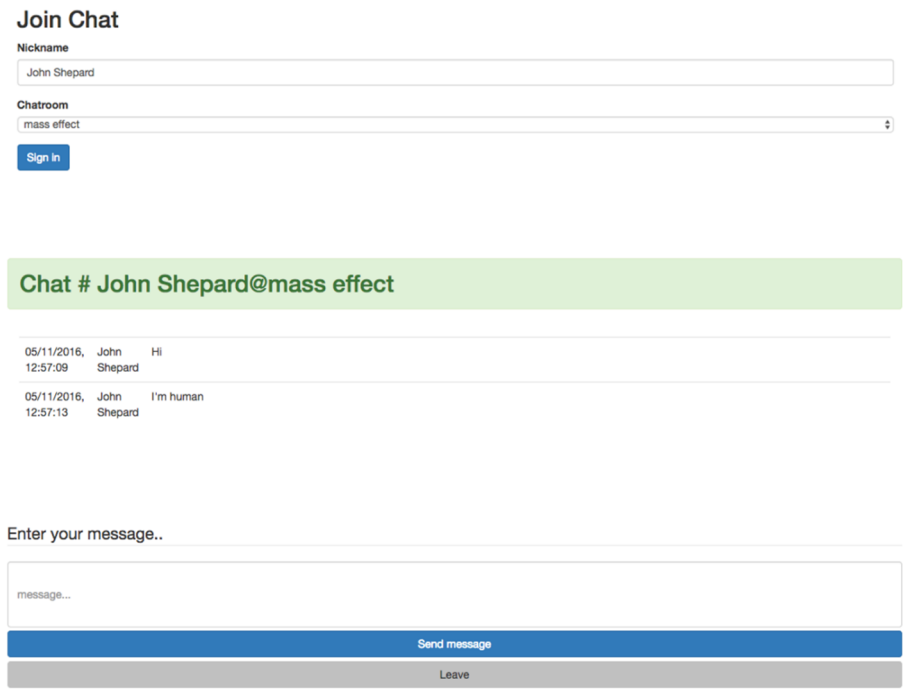
***OneWay***

Cabe señalar que al haber declarado en la API este método como *OneWay*, no se asegura la entrega y tampoco se espera respuesta (*Task<Void>*).

```
30.     Chat chat = Actor.getReference(Chat.class, "tankLobby01");
31.     ChatObserver observer = new ChatMessageObserver();
32.     chat.join(observer);
33.
34.     chat.say(“nick”, “hello”);
```

Como hemos visto en este ejemplo, en caso de querer desarrollar nuestros propios servicios, nuestro trabajo se reduce a definir la API de cada actor, implementar la lógica de los actores y exponer los servicios en el frontal de la plataforma.

La parte final es más específica sobre cómo queremos comunicarnos con el servidor. En el ejemplo proporcionado se hace uso de la extensión de Orbit para arrancar un servidor de web, que nos sirve de *proxy* gracias al puente que realiza con su *websocket*. Además, se incluye un cliente web que nos permite hacer pruebas de las diferentes funcionalidades del chat.

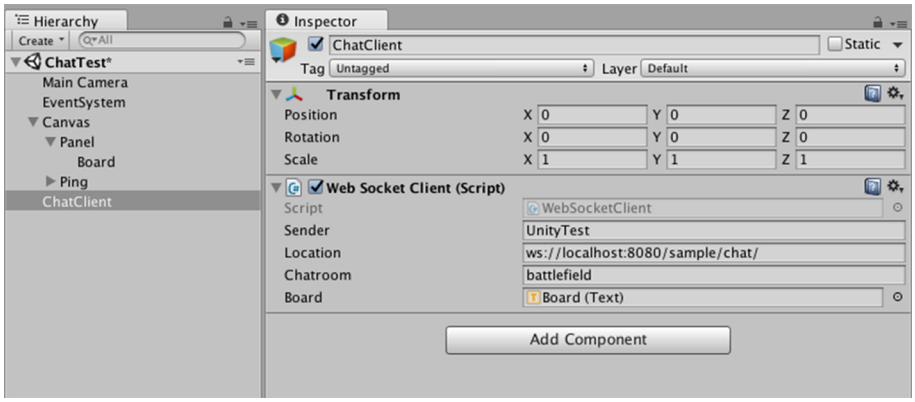


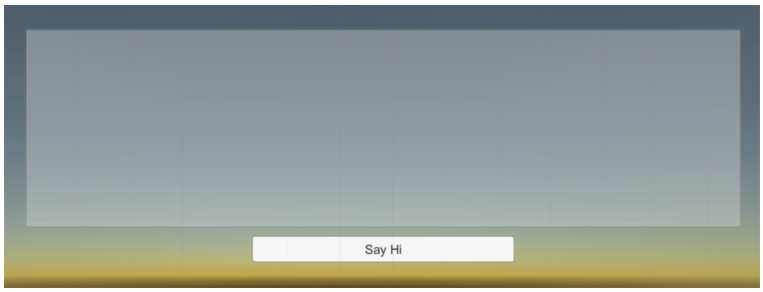

## 3.2. Lado cliente

Para esta tarea, necesitamos crear un nuevo proyecto de prueba que nos permita conectarnos y enviar y recibir mensajes desde Unity. Para conseguir esta conexión, usaremos la biblioteca *WebSocket Sharp*, que nos permite realizar de forma asíncrona la conexión, envío y recepción de mensajes.

### 3.2.1. Implementación

A partir del ejemplo proporcionado, podemos ver que la escena es muy simple y se compone básicamente dos elementos, el Canvas, con toda la interfaz de usuario, y el *GameObject ChatClient* que utiliza el script *WebSocketClient* para comunicarse con el *front-end* de Orbit.






**Reto 1**: Realizad pruebas usando diversos clientes web y clientes generados a partir del proyecto Unity.

**Reto 2**: Añadid la funcionalidad de enviar mensajes escritos por el jugador y que este pueda cambiar su *nickname*.

A continuación, debemos realizar los pasos necesarios para poder integrar el cliente con el servidor de *WebSocket* que se encuentra en el ejemplo del chat de Orbit. Comenzaremos definiendo las estructura de datos a intercambiar:

```
1.      [Serializable]
2.      public class ChatMessageDto {
3.          public string sender;
4.          public string message;
5.          public string received;
6.      }
7.
8.      [Serializable]
9.      public class ChatHistoryDto {
10.         public ChatMessageDto[] history;
11.     }
```

Así pues, la primera acción que realizaremos será inicializar el *WebSocket* cliente e iniciar el proceso de conexión, al arrancar el *GameObject* y definir las funciones de *callback* que se ejecutan una vez establecida la conexión, al recibir el mensaje y una vez se haya cerrado la conexión:

```
12.     void Start() {
13.         ws = new WebSocket(location + chatroom);
14.         ws.OnOpen += OnOpenHandler;
15.         ws.OnMessage += OnMessageHandler;
16.         ws.OnClose += OnCloseHandler;
17.
18.         ws.ConnectAsync();
19.     }
```

Por lo tanto, es primordial definir correctamente la recepción asíncrona de mensajes, sobre todo en este caso que tenemos dos paquetes distintos para recibir, el histórico de mensajes (primer mensaje que recibimos) y los mensajes del resto de jugadores, donde utilizaremos un sistema de cola para que sea la función *Update()* la encargada de renderizar el resultado:

```
20.     public Queue<ChatMessageDto> messages =
21.         new Queue<ChatMessageDto>();
22.
23.     void OnMessageHandler(object sender,
24.         MessageEventArgs e) {
25.         string header = GetHeader(e.Data);
26.         if (header.Equals("history")) {
27.             ChatHistoryDto hist =
28.                 JsonUtility.FromJson<ChatHistoryDto>(e.Data);
29.             foreach (ChatMessageDto msg in hist.history) {
30.                 messages.Enqueue(msg);
31.             }
32.         } else {
33.          ChatMessageDto msg =
34.              JsonUtility.FromJson<ChatMessageDto>(e.Data);
35.          messages.Enqueue(msg);
36.         }
37.     }
38.
39.     void Update() {
40.         while (messages.Count > 0) {
41.             ChatMessageDto msg = messages.Dequeue();
42.             board.text += msg.received+
43.                 " "+msg.sender+"  "+msg.message+"\n";
44.         }
45.     }
```

**Reto 3**: Haced que los mensajes de cada jugador sean de un color diferente para poder diferenciarlos más fácilmente.

Como paso final, se implementa una función explícita para enviar algún tipo de información, básicamente imitando un *ping*, que permite que el jugador pueda enviar un mensaje preestablecido mediante un botón:

```
46.     public void Ping() {
47.         ChatMessageDto msg = new ChatMessageDto ();
48.         msg.sender = "UnityTest";
49.         msg.message = "Ping";
50.         ws.SendAsync(msgJson, OnSendComplete);
51.     }
```

**Reto 4**: Evitad que se puedan enviar mensajes sin estar conectado para solventar errores de conexión.

Llegados a este punto, podéis generar el juego con Unity y ejecutar varias instancias en vuestra máquina, o para que las pruebas sean más reales, desde diferentes máquinas (físicas o virtuales), incluso separando el *frontend* y el *backend* de Orbit y cada cliente en una máquina cada uno.

**Reto 5**: ¿Cómo añadiríais este cliente de chat al *Lobby* del proyecto *Tank!* LAN?

**Reto 6**: ¿Qué modificarías del nuevo *Lobby* para poder usarlo en red vía Internet?

**Reto 7**: ¿De qué forma podríamos aprovechar el Chat para que los jugadores se enteraran de las nuevas sesiones que abren el resto de jugadores?

## 3.3. Soluciones a los retos

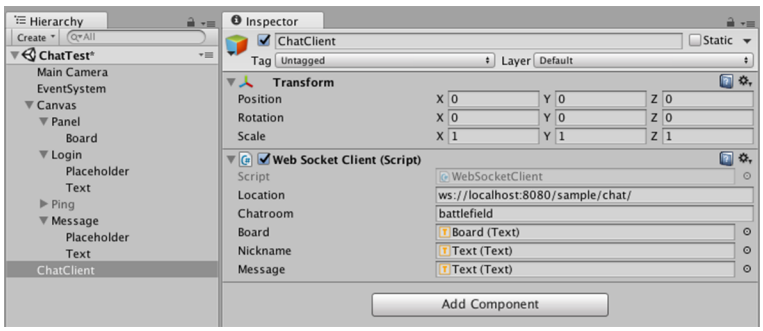
### 1) Reto 1

Para este reto debemos arrancar la plataforma Orbit con la implementación del servicio de chat, tanto el *backend* como el *frontend*, y a su vez, diferentes instancias de navegador y del juego como clientes de la plataforma. Por lo tanto, el primer paso es generar el juego como hemos realizado en módulos anteriores.

Una vez tengamos diversos clientes atacando al *frontend* del chat, entramos en la misma sala, y realizamos diferentes comentarios por cada uno de ellos y a poder ser lo visualizamos todo al mismo tiempo para ver cómo los cambios suceden de forma instantánea.

### 2) Reto 2

Para completar este reto debemos modificar la interfaz de usuario para que permita a los jugadores introducir texto, tanto para el *nickname* como para el mensaje que quieran enviar.





A continuación deberemos modificar el *script* de *WebSocketClient* para que envíe esta información, sustituyendo los campos estáticos anteriores:

```
1.      public void SendText() {
2.          ChatMessageDto msg = new ChatMessageDto ();
3.          msg.sender = nickname.text;
4.          msg.message = message.text;
5.          string msgJson = JsonUtility.ToJson(msg);
6.
7.          ws.SendAsync(msgJson, OnSendComplete);
8.      }
```

### 3) Reto 3

Para implementar la funcionalidad de mostrar de un color diferente los mensajes de cada jugador, necesitaremos enviar los mensajes codificados con los *tags* adecuados para enriquecer el texto.

En este caso concreto se requiere el *tag* especial color, y este lo usaremos para decorar los mensajes antes de enviarlos por consola:

```
1.      <color="#80000">UnityTest  Hello Wolrd!</color>
```

Por último, para representar a cada jugador con un color distinto, una opción sería generarlo aleatoriamente y asignarlo al jugador para toda la sesión.

Para este ejemplo vamos a optar por una mucho más rápida y sencilla, en la que calcularemos el color a partir del *nickname* del jugador.

```
2.      private string DecorateColor(string user, string msg) {
3.          string colorCode = GetColor(user);
4.          return "<color=" + colorCode + ">" + msg + "</color>";
5.      }
```

Teniendo en cuenta que necesitamos un algoritmo de *hash* que nos retorne los seis dígitos hexadecimales que representan la codificación RGB del color, podemos optar por un método del siguiente estilo: con cálculo explícito del *hash* u optar por un algoritmo de *message digest* conocido y usable mediante herramientas de criptología, como MD5 o SHA1:

```
6.      private string GetColor(string code) {
7.          int hash = 0;
8.          for (int i = 0; i < code.Length; i++) {
9.              hash = code[i] + ((hash << 5) - hash);
10.         }
11.
12.         string colour = "#";
13.         private int port = 11000;
14.         for (int i = 0; i < 3; i++) {
15.             int value = (hash >> (i * 8)) & 0xFF;
16.             string hex = "00"+ value.ToString ("X");
17.             colour += hex.Substring(hex.Length-2);
18.         }
19.
20.         return colour;
21.     }
```

Con estos sencillos cambios podemos comprobar al ejecutar de nuevo el ejemplo que ahora la representación de los mensajes pasa a tener los colores diferenciadores:


**4) Reto 4**

Para evitar el envío de mensajes sin estar conectado y solventar posibles errores de conexión, así como mantener un control del estado del cliente, una forma sencilla es hacerlo mediante una variable booleana:

```
1.      private bool joined;
```

Gracias a ella podemos evitar que se hagan intentos de actualización o envíos comprobando su estado justo al inicio de estos métodos:

```
2.      public void Update () {
3.          if (!joined)
4.              return;
5.      public void Ping() {
6.          if (!joined)
7.              return;
```

Para establecer su estado debemos seguir los métodos *callback* propios del *WebSocket*, donde por ejemplo damos por establecida la conexión al recibir el primer mensaje, que, como hemos visto, en este caso es el histórico, y por tanto no dejamos enviar mensajes hasta que no tenemos actualizado nuestro historial.

```
8.      void OnMessageHandler(object sender, MessageEventArgs e) {
9.          joined = true;
```

Finalmente, si se detecta que la conexión se ha cerrado, entonces damos por concluida la sesión en nuestro control.

```
10.     void OnCloseHandler(object sender, CloseEventArgs e) {
11.         joined = false;
```

**5) Reto 5**

Una forma sencilla de integrar proyectos con Unity es la de exportar/importar *assets* propios. Para ello debemos ir al menú correspondiente en el editor del proyecto *Tanks! Chat* que tenemos hasta ahora y realizar una exportación de todos los *assets* hacia un fichero externo.

Una vez guardado el fichero del *asset* en nuestra máquina, abrimos el proyecto destino (*Tanks! LAN*) e importamos este *asset* como si se tratara de un *asset* bajado de la tienda. Unity nos pedirá seleccionar los recursos que queremos importar, que en este caso serán todos, y ya tendremos incorporado el chat en nuestro otro proyecto. Podemos ejecutarlo abriendo la escena importada para verificar que el funcionamiento sea correcto.

A continuación, abrimos el proyecto de *Tanks! LAN* y duplicamos la escena *Lobby*, para crear otra nueva que se llame *NewLobby*. Esta escena nueva es donde debemos integrar el proyecto *Tanks Chat*.

Abrimos las dos escenas a la vez y movemos todos los objetos que necesitamos de la escena importada, hacia la nueva que acabamos de copiar. Cerramos la escena antigua y continuamos trabajando en la nueva para acabar de ajustar temas de interfaz gráfica reubicando el *canvas* y probando en la ejecución que todo es correcto.

**6) Reto 6**

Para convertir el juego en conexión remota debemos permitir, como hemos visto en otros retos, que el jugador se una a una partida ya creada, pero para ello indique la dirección IP del *host*.

El primer paso es modificar el *script* del *LobbyMenu*, donde podemos añadir una nueva función llamada *JoinIpGame()* para recuperar de un campo de entrada la IP destino (por ejemplo, *ipAddr*):

```
1.      public void JoinIpGame() {
2.          Network.Connect(ipAddr.text, Network.player.port);
3.          NetworkManager.singleton.StartClient();
4.      }
```

Para finalizar, pasamos a retocar la interfaz gráfica y añadimos ese campo de entrada de la IP destino que ya hemos referenciado en el *script*. Llegados a este punto, solo nos restará un nuevo botón (*Join IP Game*) al lado del campo de entrada de la IP que permita accionar la función *JoinIpGame()* para unirse a la partida.

**Acceso bajo NAT**

Recordad que los puertos en el enrutador del *host* deben estar abiertos en la configuración NAT para poder ser accedidos desde fuera.

### 7) Reto 7

**Recuperación de IPs**

Recordemos que para poder recuperar la IP externa del jugador podemos usar la función de la API de Network.

Volvemos de nuevo a la escena *NewLobby* del proyecto *Tanks! LAN* donde hemos integrado el ejemplo del chat para poder lanzar un mensaje automático cada vez que un jugador haya creado una nueva sesión de juego.

Por lo tanto, dado que para poder crear la sesión se debe llamar a la función *CreateGame()*, vamos a añadir el código necesario para poder enviar mensajes desde este *script*.

Primero vamos a necesitar acceder al *GameObject* del *ChatClient*. Una forma fácil de hacerlo es mediante la API de Unity:

```
1.      GameObject client = GameObject.Find("ChatClient");
```

Con la referencia al *GameObject* y teniendo la IP destino ya guardada, lanzamos, desde el final de la función *CreateGame()*, una llamada a otra función del *script* que nos permita enviar el mensaje genérico de invitación vía *websockets* a la sala de chat:

```
2.      public void AnnounceGame() {
3.          String myIp = Network.player.externalIP;
4.          GameObject client = GameObject.Find("ChatClient");
5.          WebSocketClient ws =
6.              client.GetComponent<WebSocketClient>();
7.
8.          ws.message.text = "My game is open, join me! "+myIp;
9.          ws.SendText ();
10.     }
```

Finalmente, si probamos el ejemplo completo con todas estas modificaciones, deberemos ser capaces de ver los mensajes de anuncio de nueva partida y usar esa IP para acceder a la partida en remoto vía Internet.

# Resumen

En este último módulo hemos podido repasar lo que son hoy en día los servicios indispensables para cualquier juego, aunque no se trate ni de un juego en línea ni multijugador. Estos servicios son propios de plataformas muy populares, como el ejemplo de Steam que hemos estudiado. Pero estos servicios se encuentran obviamente en el lado servidor de cada plataforma. Alternativamente podemos encontrar proyectos como Sugar, que nos permiten utilizar sus servicios ya implementados en nuestros desarrollos, independientemente de la plataforma.

**Enlace recomendado**

Para más información sobre el proyecto Sugar podéis consultar su web oficial: http://sugarengine.org.

Para entender cómo trabajan estos servicios, hemos entrado en el paradigma de la computación en la nube, que hoy en día es la forma más recomendada para realizar este tipo de tareas remotas. Como hemos visto, existen diferentes tipos de nube, cada una con sus ventajas e inconvenientes. El coste es uno de ellos, aunque que no sea realmente un problema nuevo, dado que antes, con soluciones de servidor tradicionales, el problema era más injusto, pues se facturaba por infraestructura total y no por consumo real.

Existen diferentes alternativas a los servicios en la nube oficiales para Unity, como el propio ofrecido por ellos, u otros como Photon Cloud. Son servicios de pago que nos ofrecen una forma sencilla de desplegar nuestro juego realizando el mínimo de cambios posible. Alternativamente, podemos buscar otros *assets* que nos permitan integrar nuestro juego en una plataforma concreta o específica del propio *asset* y dotarlo de los servidores dedicados.

**Ejemplo**

Un ejemplo de *asset* sería el Master Server Kit: https://www.assetstore.unity3d.com/#!/content/71604.

Finalmente hemos recuperado el ejemplo que hemos trabajado en Unity en los módulos anteriores, *Tanks!*, para integrarlo en una plataforma real como es Orbit. Este proyecto de libre distribución creado por Bioware nos permite crear todo tipo de servicios descentralizados y escalables en la nube, que podemos hospedar en plataformas conocidas como las que proporciona Amazon, o encargarnos nosotros de hospedarlo donde creamos conveniente como *cloud* privado.

Gracias a ello, podemos arrancar en local el servicio de chat y realizar las pruebas directamente contra cualquier videojuego que se integre en él. Así pues, después de realizar dicha integración utilizando la tecnología de *WebSockets*, hemos podido comprobar el funcionamiento de la plataforma completa desde el *backend*, pasando por el *frontend* y llegando a los clientes.

# Bibliografía

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

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

**Coulouris, George; Dollimore, Jean; Kindberg, Tim** (2005). «Distributed systems: concepts and design». En: *International computer science series* (vol. 4). Reading, MA: Addison Wesley.

**Jiménez Rodríguez, J.; Jiménez Díaz, G.; Díaz Agudo, B.** (2011). «Match-making and case-based recommendations». En: *Workshop on Case-Based Reasoning for Computer Games*, XIX Conferencia Internacional sobre *Case Based Reasoning*.

**Glickman, M. E.** (1999). «Parameter estimation in large dynamic paired comparison experiments». *Applied Statistics* (vol. 48, págs. 377-394).

**Herbrich, R.; Minka, T.; Graepel, T.** (2007). «TrueSkill(TM): a bayesian skill rating system». *Advances in Neural Information Processing Systems* (vol. 20, págs. 569-576).

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

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