ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Curso Académico 2023/2024

Trabajo Fin de Grado

CREACIÓN DE UN PROTOTIPO DE VIDEOJUEGO DE DISPAROS EN PRIMERA PERSONA MULTIJUGADOR EN LÍNEA

Autor: Daniel Jiménez Morales

Directores: Julio Guillén García

Universidad Rey Juan Carlos | Escuela Técnica Superior Ingeniería Informática

Quiero expresar mi profunda gratitud, en primer lugar, a mi tutor Julio Guillén por su constante apoyo a lo largo de este proyecto. Su desempeño fue fundamental a la hora de completar este trabajo.

Asimismo, agradezco sinceramente a mi familia, amigos y compañeros por su constante ánimo y comprensión durante este periodo de esfuerzo y dedicación.

Finalmente, mi más sincero agradecimiento a todos aquellos profesionales de la industria, en particular a Jay Mattis y Shaun Sheppard por compartir su invaluable conocimiento y experiencia los cuales contribuyeron significativamente a la realización de este trabajo.

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería Informática

Universidad Rey Juan Carlos | Escuela Técnica Superior Ingeniería Informática

Resumen

En este trabajo se diseña e implementa todo el apartado multijugador de un prototipo de videojuego de disparos en primera persona utilizando el motor *Unity*. Además, se emplea una arquitectura cliente-servidor junto con la técnica de interpolación de *snapshots* para sincronizar los distintos clientes con el servidor.

Como estudio teórico previo al desarrollo de este prototipo, se ha realizado una investigación exhaustiva que abarca los inicios de los videojuegos multijugador, el análisis numerosos casos de estudio relevantes y un estudio detallado de una gran variedad de conceptos, estrategias y algoritmos enfocados en resolver los desafíos inherentes a este tipo de juegos.

Finalmente, para validar y evaluar los resultados obtenidos de esta implementación, se han llevado a cabo diversos tipos de *tests* probando el prototipo. Estos *tests* incluyeron una fase de *beta testing* con usuarios reales y la demostraciones gráficas que ilustran el correcto funcionamiento y los beneficios clave de algunos de los sistemas más importantes implementados en este proyecto.

Palabras Clave:

Netcode

Multijugador en línea

Interpolación de *snapshots*

Videojuego multijugador

Cliente-servidor

Universidad Rey Juan Carlos | Escuela Técnica Superior Ingeniería Informática

Abstract

This work involves designing and implementing the entire multiplayer section of a first-person shooter video game prototype using the *Unity* engine. Furthermore, it employs a client-server architecture along with *snapshot* interpolation technique to synchronize different clients with the server.

As a theoretical study preceding the development of this prototype, an extensive investigation has been conducted. This investigation covers the origins of multiplayer video games, the analysis of numerous relevant case studies, and a detailed study of a wide array of concepts, strategies, and algorithms focused on addressing the inherent challenges in this type of gaming.

Finally, to validate and assess the results obtained from this implementation, various types of tests have been conducted on the prototype. These tests encompassed a beta testing phase involving real users and graphical demonstrations illustrating the proper functioning and key benefits of some of the most important systems implemented in this project.

Keywords:

Netcode

Online multiplayer

Snapshot interpolation

Multiplayer videogame

Client-Server

Índice de contenidos

Índice de ilustraciones

Glosario

First Person Shooter (FPS)	Siglas de *First Person Shooter*, traducido al español como "Juego de Disparos en Primera Persona", es un tipo de videojuego de disparos donde el jugador ve el mundo desde la perspectiva del personaje, es decir, a través de sus ojos.
Jitter	En inglés, fluctuación de la latencia de la red.
Offline	Sin conexión.
Online	En línea.
Internet Protocol (IP)	En inglés, Protocolo de Internet, el cual es un conjunto de reglas y estándares que permiten la comunicación entre distintos dispositivos a través de internet o una red local.
Dirección IP	Dirección única que identifica a un dispositivo en internet o en una red local.
User Datagram Protocol (UDP)	En inglés, Protocolo de datagramas de usuario, es un protocolo de comunicación no fiable basado en la transmisión de datagramas.
Loopback/ Localhost	Interfaz de red virtual que usando una dirección IP, especial para redirigir el tráfico de red hacia el dispositivo de origen de nuevo sin pasar por una red externa.
Netcode	En inglés, código de red, encargado de gestionar el funcionamiento de la red de un videojuego multijugador.
Testing	En inglés, proceso de pruebas, es una fase del desarrollo de un videojuego en la que se comprueba el correcto funcionamiento de un aspecto, sistema, componente o conjunto de estos.
Test	En inglés, prueba de la fase de *testing,* tiene como objetivo evaluar y validar el comportamiento, la funcionalidad y/o la calidad de un aspecto, sistema o componente de un videojuego.
Dumb client	En inglés, cliente tonto, se dice de aquellos clientes que únicamente actualizan el estado del juego en base a los mensajes recibidos por parte del servidor.
Log	En inglés, registro secuencial y cronológico de las distintas operaciones realizadas en un sistema informático.
Input	En inglés, entrada, es un conjunto de datos compuestos por teclas y/o botones presionados por un jugador en un videojuego durante un instante de tiempo determinado.
Peer To Peer (P2P)	En inglés, red entre pares, es una arquitectura de red.

Peer	En inglés, par, es un nodo de la arquitectura Peer To Peer.
Partial update	En inglés, actualización parcial del estado del juego.
Ghost	En inglés, fantasma, es una copia de los objetos sincronizados a través de la red en un cliente.
Remote Procedure Call (RPC)	En inglés, Llamada a Procedimiento Remoto, permite a un programa ejecutar código en remoto facilitando la comunicación entre sistemas distribuidos y comúnmente usado en arquitecturas cliente-servidor.
Snapshot	En inglés, instantánea, es una estructura en la que se almacena la totalidad del estado de juego en un instante de tiempo determinado.
Delta-snapshot	En inglés, instantánea delta, es una estructura en la que se almacenan únicamente aquellos datos que han sufrido algún tipo de modificiación o han sido añadidos respecto a la última *snapshot*.
Game engine	En inglés, motor de videojuegos.
Buffer	En inglés, búfer, es un espacio de memoria destinado a almacenar datos de forma temporal.
Local Area Network (LAN)	En inglés, red de área local.
Tick	En inglés, intervalo regular de tiempo, en el que un servidor o cliente realiza actualizaciones y procesa eventos.
Packet server	En inglés, servidor de paquetes, es una arquitectura tanto con elementos propios de cliente-servidor como de P2P. En esta arquitectura, un único jugador hará tanto de cliente como de servidor de *relay* redirigiendo los *inputs* de un cliente al resto que estén conectados.
Ping	Intervalo de tiempo desde que el jugador ejecuta una acción, es procesada por el servidor y llega de vuelta al cliente para ser renderizada y percibida por el jugador.
Servidor de relay	Servidor que actúa de intermediario entre dos o más dispositivos para facilitar su comunicación. Su función principal es retransmitir los paquetes entre los distintos dispositivos.
Bit	En inglés, dígito binario, Unidad mínima de información la cual corresponde a un dígito del sistema de numeración binario.
Bit packing	En inglés, empaquetamiento de *bits*, es una forma de compresión que elimina aquellos *bits* que no son relevantes y empaqueta los *bits* restantes.
Byte	En inglés, octeto de dígitos binarios, es la unidad de información compuesta por ocho *bits*.
Reliable message	En inglés, mensaje fiable, es aquel que se garantiza que va a llegar a su destino de manera segura.

Unreliable message	En inglés, mensaje no fiable, es aquel que no se garantiza que llegue a su destino.
Acknowledgement (ACK)	En inglés, reconocimiento o acuse de recibo, son mensajes que tienen la finalidad de informar que un paquete de datos ha sido recibido correctamente.
Run-Length encoding (RLE)	En inglés, codificación de longitud de ejecución, Algoritmo de compresión de datos sin pérdida basado en la repetición de valores consecutivos dentro de una secuencia de datos.
Run	Repetición de valores consecutivos en el algoritmo de compresión RLE.
String	En inglés, cadena de caracteres.
Deterministic lockstep	En inglés, coordinación determinista, es una técnica de sincronización en la que las distintas máquinas sincronizan los *inputs* del resto de los jugadores y tras ello simulan el estado de juego de manera determinista.
Round-Trip time (RTT)	En inglés, tiempo de ida y vuelta, es el tiempo que tarda un paquete de datos en viajar desde el origen hasta llegar al destino y luego regresar al punto de origen.
Host	En inglés, anfitrión, se dice de cualquier dispositivo que posee una dirección IP y puede ser identificado en una red. También, se dice de un nodo que actúa tanto como un cliente como un servidor.
Stateful	En inglés, con estado, es una propiedad de un sistema que tiene en cuenta el estado pasado a la hora de llevar a cabo el cálculo del nuevo estado.
Software	En inglés, conjunto de componentes lógicos necesarios de un sistema informático que le permiten realizar tareas específicas.
Hardware	En inglés, conjunto de componentes físicos que componen la parte física de un sistema informático.
Bug	En inglés, error de software, el cual provoca un comportamiento inesperado o no deseado.
Glitch	En inglés, error de software de naturaleza visual o sonora.
Extensible Markup Language (XML)	En inglés, lenguaje de marcado extensible.
Debug	En inglés, depuración, es el proceso de identificar y corregir errores de programación. También es conocido por el término Debugging.
Gameplay	En inglés, jugabilidad, proceso que incluye todas las acciones y decisiones que un jugador realiza mientras participa en el juego.
Thread-safe	En inglés, hilo seguro, se dice de aquellas operaciones que pueden ser ejecutadas de forma simultánea por varios hilos de ejecución sin comprometer la integridad de los datos.

Bot	En inglés, robot, se dice de toda entidad controlada por inteligencia artificial.
Payload	En inglés, carga útil, conjunto de datos útiles transmitidos en un paquete, excluyendo otros datos o metadatos como la cabecera de este.
Maximum Transmission Unit (MTU)	En inglés, unidad máxima de transferencia, es un término que determina el tamaño máximo de un paquete de datos, expresado en bytes, que puede ser transferido a través de la red.
Internet Protocol Version 4 (IPv4)	En inglés, versión 4 del protocolo de internet.
Internet Protocol Version 6 (IPv6)	En inglés, versión 6 del protocolo de internet.
Bufferbloat	En inglés, hinchamiento de búfer, es un problema de la red generado cuando un nodo posee un *buffer* excesivamente grande para almacenar paquetes, lo que resulta en una retención prolongada de paquetes en dicho *buffer*. Este fenómeno causa problemas como alta latencia y/o *jitter*.
Active Queue Management (AQM)	En inglés, Políticas de Manejo Activo de Cola, son algoritmos encargados de descartar paquetes de red dentro de un *buffer* para evitar que este se llene y/o se genere el fenómeno de *Bufferbloat*.
Broadcast	En inglés, difusión, es una forma de comunicación en la que un mensaje es enviado desde un único emisor a todos los dispositivos conectados al emisor.
Cluster	En inglés, grupo, conjunto de servidores.
Server mitigation	En inglés, mitigación del servidor, proceso mediante el cual se transfiere la autoridad del servidor de un cliente a otro.
Time warp	En inglés, distorsión temporal, técnica de compensación de la latencia en la que se retrocede el estado de juego a un instante en el pasado en el que se produjo una acción con el objetivo de aplicar dicha acción, compensando así la latencia, y volviendo a avanzar el estado de juego hasta la actualidad.
Sticky targets	En inglés, objetivos pegajosos, es una técnica de control de asistencia en la cual se reduce la cantidad de movimiento del cursor cuando este se encuentra cerca de un objetivo.
Flag	En inglés, bandera, es una variable comúnmente utilizada para representar un estado específico. En la mayoría de los casos, esta variable es un booleano con los estados de Verdadero o Falso.
Dead reckoning	En inglés, navegación a estima, también conocido como extrapolación, es una técnica de compensación de la latencia que tiene como objetivo

	predecir estados futuros de objetos controlados por otros jugadores asumiendo que sus comportamientos actuales seguirán invariables.
Exponential smoothing	En inglés, suavizado exponencial, es una técnica que tiene el objetivo de ir corrigiendo de forma suave un error a lo largo del tiempo aplicando un porcentaje de la corrección en cada fotograma.
Projective velocity blending	En inglés, mezcla de velocidades proyectivasraducido como mezcla de velocidades proyectadas, es un algoritmo capaz de realizar los proceso de extrapolación e interpolación de forma simultánea.
Spline	Curva diferenciable definida en porciones mediante el uso de polinomios.
Collider	Traducido como colisionador, es un componente utilizado para detectar las colisiones entre distintos objetos de un mundo virtual.
Raycast	Línea recta cuyo objetivo es buscar los posibles puntos de intersección entre dicha línea y el resto de *colliders*.
Sphere cast	Variante del *raycast* en la que, a diferencia del *raycast* donde se utiliza una línea recta a lo largo de una dirección, en el *sphere cast* se utiliza una esfera a lo largo de una dirección.
Hitscan	Tipo de arma en la que las balas viajan a una velocidad infinita, permitiendo saber si estas han colisionado con algún objeto en el momento en el que son disparadas.
Hit registration	Proceso mediante el cual se comprueba si una bala ha colisionado con algún objeto de un mundo virtual.
Latency concealment	Traducido como ocultación de la latencia, es una técnica de compensación de la latencia la cual añade efectos tanto visuales como sonoros con el objetivo de aumentar la responsividad del juego.
Hitmarker	Elemento visual y/o sonoro que indica que un disparo ha alcanzado con éxito a un objetivo.
Server-side rewind	Técnica de *Time warp* realizada en el servidor para replicar el estado de juego de tal forma que este sea idéntico al del cliente en el momento que realizó el disparo durante el proceso de *Hit registration*.
Shot behind cover	Situación en la cual un jugador es dañado por una bala inmediatamente después de haberse puesto a cubierto tras una cobertura.
Conditional lag compensation	Traducido como compensación de lag condicional, es una técnica que tiene como objetivo reducir el número de situaciones *Shot behind cover* limitando la cantidad de tiempo atrás que el servidor puede retroceder el estado de juego en la técnica del *Server-side rewind.*
Advance lag compensation	Traducido como compensación de lag avanzada, es un técnica que tiene como objetivo reducir el número de situaciones *Shot behind*

	cover haciendo que la víctima corrobore el resultado de un disparo en base a su versión local del estado de juego.
Observer	Patrón de diseño de software que define una relación del tipo uno a muchos entre objetos, de forma que cuando uno de los objetos cambia de estado, todos sus observadores son notificados de forma automática.
Nimbus	Representa el área en el que un objeto puede ser observado, escuchado, leído, entre otros, por un objeto observador en un medio específico.
Zoom in/out	Términos utilizados para referirse al acercamiento (*Zoom in*) o alejamiento (*Zoom out*) de la imagen.
Blueprints	Sistema visual de scripting basado en nodos, propio del motor de videojuegos Unreal Engine, que permite a los desarrolladores y diseñadores crear la lógica del juego sin necesidad de usar la programación tradicional.
Matchmaking	Proceso de emparejar a jugadores de forma apropiada, en base a una serie de reglas de emparejamiento, con el objetivo de que puedan competir o colaborar en una experiencia de juego.
Array	Estructura de datos que contiene elementos del mismo tipo en una secuencia contigua de memoria.
Sprite	Imagen bidimensional utilizada como elemento visual en un entorno gráfico.
Beta testing	Fase de pruebas de un videojuego durante la cual se pone a disposición de un grupo limitado de usuarios externos una versión del videojuego, conocida como versión beta, para que la utilicen y proporcionen retroalimentación.
Input lag	Cantidad de tiempo de retraso entre que un jugador presiona una tecla o un botón hasta que la acción correspondiente se visualiza en pantalla.
Test de PlayMode	Prueba automatizada la cual se ejecuta durante el modo de reproducción del editor de Unity Engine. La finalidad de estas pruebas es evaluar el comportamiento de un proyecto en tiempo de ejecución.
Programmed Logic Automated Teaching Operations (PLATO)	Plataforma educativa desarrollada por Donald Bitzer dentro de la Universidad de Illinois en 1960.
Mainframe	Servidor central de la plataforma PLATO.
Telnet	Protocolo de red el cual permite la comunicación de un varios nodos mediante el uso de una interfaz basada en texto.

Internetwork Packet Exchange (IPX)	Protocolo de red desarrollado por Novell en la década de los 80.

Capítulo 1

Introducción

En la sección de Introducción, se describe el problema abordado en el desarrollo de este prototipo multijugador en línea así como la motivación y los objetivos. La descripción del problema se centra en los desafíos técnicos, presentes en el desarrollo de todo juego multijugador en línea. La motivación se fundamenta en la creciente demanda y popularidad de los juegos multijugador, así como en el la falta de recursos tanto teóricos como prácticos dentro de este campo. Los objetivos se plantean con claridad, enfocándose en los aspectos multijugador capaces de dar solución a desafíos específicos en este tipo de experiencias en línea. Esta sección establece el contexto y los propósitos fundamentales que guían el desarrollo de este prototipo.

1.1 Descripción del problema

El desarrollo de videojuegos multijugador en línea ha experimentado un crecimiento exponencial en la última década, entre los que destacan los videojuegos multijugador en línea del género *First Person Shooter* (FPS, en inglés, Juego de disparos en primera persona). Estos juegos ofrecen experiencias altamente competitivas que involucran a jugadores de todo el mundo. Sin embargo, uno de los desafíos más significativos en la creación de un FPS multijugador en línea radica en las diferentes condiciones en la red y su impacto en la jugabilidad.

Debido a los diferentes problemas de la red como la latencia, la pérdida de paquetes o el *jitter* (en inglés, fluctuación de la latencia de la red), existe una probabilidad de que los paquetes lleguen tarde, desordenados o simplemente no lleguen al destino. Es por lo que los programadores encargados de desarrollar el apartado multijugador de estos juegos deben de saber tratar estos fenómenos y de alguna forma minimizarlos para que la experiencia final del jugador no se vea gravemente afectada.

1.2 Motivación

El presente Trabajo de Fin de Grado se originó a partir de una profunda inquietud por abordar una carencia evidente en la literatura técnica relacionada con los aspectos multijugador de los videojuegos, en particular, aquellos que se centran en el género de FPS. La motivación principal para emprender este proyecto radica en la constatación de que, si bien existen

algunos recursos disponibles que abordan estos conceptos, la mayoría de ellos tienden a tratarlos de manera superficial, sin profundizar demasiado, y teórica sin mostrar una posible forma de implementación práctica.

La falta de recursos técnicos específicos en el ámbito del multijugador en videojuegos deja a los desarrolladores y entusiastas, como el autor de este documento, con una sensación de incertidumbre.

Por todo lo anterior, este trabajo se origina, como un esfuerzo por llenar ese vacío en el conocimiento técnico. No solo se busca explorar en profundidad los conceptos clave relacionados con el multijugador en juegos FPS, sino también proporcionar una guía práctica sobre la implementación de sistemas específicos diseñados para minimizar el impacto de las condiciones de red en la experiencia final del juego. La intención es que este trabajo sirva como un recurso valioso y práctico tanto para desarrolladores en busca de soluciones concretas como para aquellos interesados en comprender mejor los desafíos técnicos que enfrenta la industria de los videojuegos en línea.

1.3 Objetivos

Objetivo general

Desarrollar un prototipo funcional de un FPS online multijugador que garantice una experiencia en red fluida y atractiva para los jugadores.

Objetivos específicos

Diseñar e implementar las mecánicas de juego en un entorno *offline*, es decir, que tiene lugar sin una conexión a la red, para establecer una base sólida para la jugabilidad.

Implementar técnicas de compensación de latencia tanto en el cliente como en el servidor con la finalidad de mejorar la experiencia en línea de los jugadores.

Proporcionar soporte multijugador que permita a los jugadores jugar tanto a través de una conexión a internet, como a través de una conexión *loopback*, es decir, usando una dirección *Internet Protocol* (IP, por sus siglas en inglés) especial para redirigir el tráfico de red hacia ellos mismos sin pasar por una red externa.

Realizar pruebas exhaustivas de jugabilidad en línea para evaluar la efectividad de las técnicas multijugador implementadas, obtener retroalimentación sobre la experiencia de juego en línea por parte de los jugadores y ajustar la implementación según sea necesario.

1.4 Estructura de la memoria

A lo largo de esta memoria se abordarán temas directamente relacionados con el multijugador y el *netcode*, es decir, el código en red encargado de gestionar el funcionamiento de la red de un videojuego multijugador. A continuación, se proporcionará un desglose detallado de los temas que se tratarán en los apartados posteriores de este documento.

El Capítulo 3 Marco teórico estará formado por cuatro apartados principales. El primero, llamado Evolución histórica de los videojuegos multijugador, introducirá los primeros videojuegos multijugador. El segundo apartado, llamado Casos de estudio, estudiará y proporcionará una explicación detallada de la arquitectura de varios videojuegos destacados en el ámbito multijugador, resaltando sus contribuciones significativas. El tercer apartado es el de Complementos teóricos el cual abordará diversas topologías y técnicas relacionadas con el netcode, respaldadas por investigaciones académicas y otros recursos de interés. Adicionalmente, se presentarán ejemplos explicativos sobre cómo se han aplicado estos conceptos en videojuegos multijugador de gran éxito en la actualidad. Por último, este capítulo incluirá un apartado dedicado a analizar las distintas soluciones y librerías disponibles actualmente en el mercado para la creación de experiencias multijugador, proporcionando una clasificación de cada uno para evaluar tanto sus ventajas como desventajas y, a causa de ello, determinar la solución más adecuada tanto para el desarrollador como para el proyecto en cuestión. Este será el apartado de Estudio de alternativas.

El Capítulo 4 Descripción informática se centrará en explicar la implementación adoptada por el prototipo de FPS multijugador en línea, permitiendo su correcto funcionamiento. Al igual que el capítulo anterior, este se dividirá en tres apartados. El primero, llamado Análisis, presentará y enumerará los Requisitos funcionales y no funcionales del proyecto, los cuales se centrarán exclusivamente en el apartado de *netcode* excluyendo otros aspectos del desarrollo como el audio o la jugabilidad entre otros. Además, este apartado contendrá una sección en la que se detallarán las Especificaciones de los distintos programas y herramientas usados en el desarrollo de este prototipo. El siguiente apartado titulado Diseño mostrará un diagrama de componentes de la arquitectura de n*etcode* utilizada, así como varios diagramas de clases de cada uno de dichos componentes. El último apartado de este capítulo es el de Desarrollo en el que se explicarán distintos aspectos de la implementación de este prototipo como las estructuras de datos transmitidas a través de la red o ciertas técnicas mencionadas en el Capítulo 3 Marco teórico entre otros aspectos relevantes. Con el fin de facilitar la comprensión del lector, se utilizarán diagramas de secuencia en varias ocasiones.

El siguiente capítulo estará enfocado a toda la parte de los diferentes *tests* llevados a cabo a lo largo del proceso de desarrollo. Este tendrá tres apartados que abordan distintas técnicas de *testing*. Estos son el apartado de Pruebas unitarias en el que se explicarán distintas pruebas de *testing* realizadas en el prototipo. Por otro lado está el apartado de Evaluación de los algoritmos de interpolación y extrapolación en el que se ha llevado a cabo un *test* algo más complejo relacionado con los algoritmos de interpolación y extrapolación en el cliente. El apartado restante de este capítulo es el de Beta testing en la que se realizaron una serie de sesiones de juego de corta duración con jugadores voluntarios, donde se analizaron las vulnerabilidades del prototipo y se obtuvieron conclusiones basadas en los resultados observados.

Universidad Rey Juan Carlos
Escuela Técnica Superior Ingeniería Informática

Capítulo 3

Marco teórico

En este apartado se realiza un recorrido por la evolución histórica de los juegos multijugador desde sus inicios, destacando hitos clave y transformaciones significativas en su desarrollo. Se presentan casos de estudio emblemáticos que han marcado un antes y un después en el ámbito de los juegos multijugador por el uso de algún elemento innovador, analizando su impacto así como beneficios y problemáticas. Además, se tratan en detalle conceptos teóricos fundamentals en la creación de experiencias multijugador en línea, como distintas técnicas de compensación de latenciala. Esta sección proporciona un contexto histórico y teórico que enriquece la comprensión del posterior desarrollo, aportando bases sólidas para la implementación de este prototipo multijugador en línea.

3.1 Evolución histórica de los videojuegos multijugador

Desde sus modestos comienzos con juegos locales hasta la actual convergencia de entornos en línea que permiten llevar a cabo partidas a escala global, la evolución de los videojuegos multijugador ha sido un fenómeno dinámico y en constante cambio. La influencia de los avances tecnológicos ha transformado no solo la experiencia de juego, sino también el panorama económico de la industria. Hoy en día, los juegos multijugador se han convertido en una forma de entretenimiento muy lucrativa y altamente demandada. La capacidad de conectar a jugadores de todo el mundo ha creado un mercado vibrante y competitivo, donde los títulos exitosos no solo generan experiencias emocionantes, sino que también impulsan la economía de la industria del entretenimiento digital. En este contexto, se explorarán los hitos clave que han contribuido a la evolución y al éxito comercial de los videojuegos multijugador.

Se puede considerar a **Tennis For Two** [1] no solo el primer videojuego de la historia sino además, el primero que involucra la participación de dos jugadores en un entorno local de forma simultánea (multijugador local). Este título fue creado en el año 1958 por William Higginbotham dentro de un osciloscopio. En él, el mapa, el cual trataba de una pista de tenis vista desde el lateral, era renderizado mediante el uso de dos líneas. Una horizontal para definir el suelo de la pista y otra vertical justo a la mitad para definir la altura de la red. El juego consistía en la simulación de un partido de tenis virtual donde cada jugador, mediante

el uso de botones y/o perillas, podía controlar tanto la velocidad como la trayectoria de la pelota renderizada como un punto en el osciloscopio.

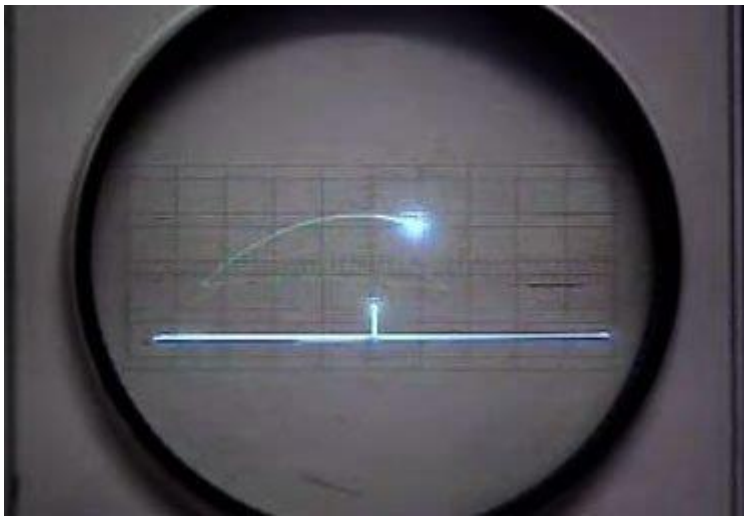


Ilustración 1 – Imagen del videojuego Tennis For Two de 1958

Unos años más tarde, concretamente en 1961, llegó Spacewar! [2] Un videojuego de combate espacial desarrollado por Steve Russell junto con una serie de estudiantes del Instituto de Tecnología de Massachusetts. En este videojuego los jugadores controlaban una nave espacial atrapada en un campo gravitacional simulado alrededor de un sol. El objetivo era el de maniobrar la nave con la finalidad de evitar colisionar con el sol mientras se intentaba destruir la nave del jugador oponente mediante disparos. Spacewar! trajo notables avances respecto a Tennis For Two, ya que este fue desarrollado en un ordenador usando un monitor para mostrar sus gráficos en lugar de un osciloscopio. Además, en el ámbito multijugador, una versión posterior de Spacewar! introdujo las partidas de red de área local o *Local Area Network* en inglés (LAN, por sus siglas en inglés), pudiéndose usar de esta forma un monitor para cada jugador.

Este hito no solo marcó avances en el ámbito de los videojuegos multijugador, sino que también hizo que los videojuegos fueran más accesibles al utilizar un ordenador como plataforma de juego. De hecho, para demostrar el potencial de los ordenadores de aquella época, el modelo PDP-1, en el cual se desarrolló Spacewar!, traía instalado el videojuego de fábrica, lo que hizo que aumentase aún más su popularidad.

Una década más tarde, en el año 1972, Atari lanzó Pong [3] considerado como el primer videojuego exitoso y popular. Pong simulaba un partido de tenis de mesa virtual en el que intervenían dos jugadores de forma simultánea. Cada jugador controlaba una pala de tenis de mesa, representada con una línea vertical en la pantalla. El objetivo de cada jugador era el de golpear una pelota, representada por un punto en la pantalla, moviendo su pala en el eje vertical. Si un jugador fallaba a la hora de devolver la pelota al campo contrario, el jugador oponente ganaba un punto. Por otra parte, el juego presenta un desafío incremental ya que la pelota se movía más rápido a medida que avanzaba el tiempo. Gracias a Pong, Atari pudo

ofrecer una experiencia multijugador a gran escala haciéndolo muy adictivo para los distintos jugadores gracias a la competitividad que este ofrecía.

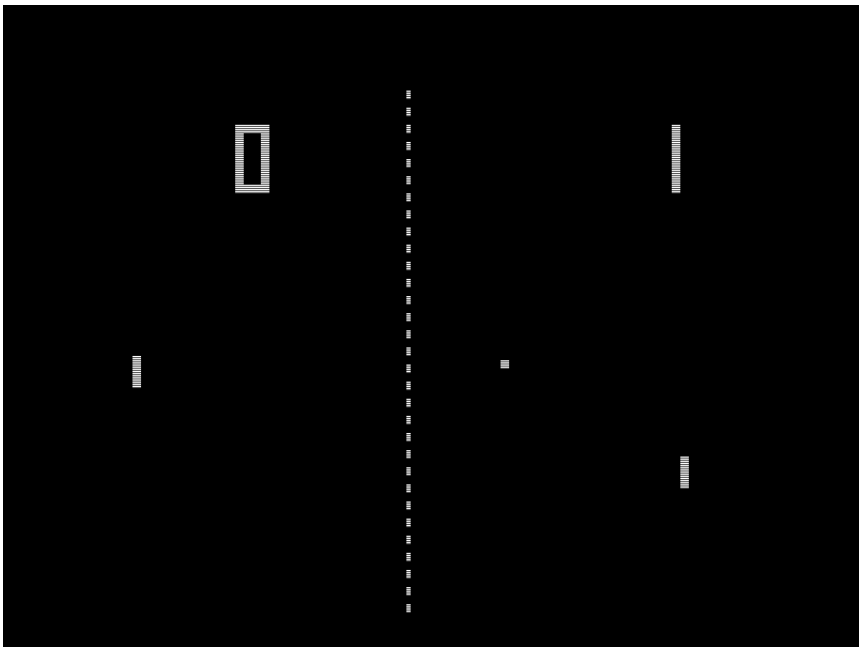


Ilustración 2 – Imagen del videojuego Pong de 1972

En 1960 se crearía PLATO (*Programmed Logic Automated Teaching Operations*, en inglés, Lógica Programada para Operaciones de Enseñanza Automatizadas) [4], una herramienta educativa, desarrollada por la Universidad de Illinois (activa hasta el año 2005). Aunque concebida principalmente con fines educativos, PLATO marcó un hito en el ámbito de los videojuegos multijugador al aprovechar la arquitectura cliente-servidor. Este sistema centralizado permitía la conexión de terminales de pantalla (conocidas como "terminales tontas" dado que no realizan tareas de cálculo o procesamiento significativas por sí mismos) a través de una red. La introducción de esta estructura posibilitó que los jugadores, al conectarse a una de estas terminales, pudieran participar en juegos multijugador en línea. El proceso implicaba que los jugadores enviaban sus comandos al servidor central, conocido como *mainframe*, que realizaba los cálculos necesarios y devolvía el estado del juego al cliente de cada jugador, que utilizaba esta información para renderizar los gráficos en pantalla. La arquitectura Cliente-Servidor supuso un avance significativo en la implementación de juegos en red, y sigue siendo ampliamente utilizada actualmente.

Las aportaciones de PLATO, dieron origen a videojuegos como los MUDs (*Multi-User Dungeon*) [5], en los que varios jugadores eran capaces de interactuar de forma simultánea mediante el uso de comandos de texto para ir tomando decisiones y acciones que alterarían el estado del mundo virtual en tiempo real. Los primeros juegos de este tipo estaban inspirados en el juegos de rol de mesa como Dragones y Mazmorras aunque con el tiempo el abanico de temáticas se fue expandiendo. Una de las principales características de los MUDs era la persistencia de los mundos. Cuando los jugadores se desconectaban del servidor, las acciones y decisiones tomadas anteriormente permanecían vigentes en el mundo del juego e incluso podían evolucionar como consecuencia de los *inputs* de otros jugadores. Esto hizo que los mundos

virtuales se sintiesen vivos y en constante evolución. Por otra parte, la arquitectura cliente-servidor de los MUDs usaba un protocolo de comunicación llamado *Telnet*, el cual permitía a los clientes conectarse a un servidor mediante una interfaz basada en texto a través de internet.

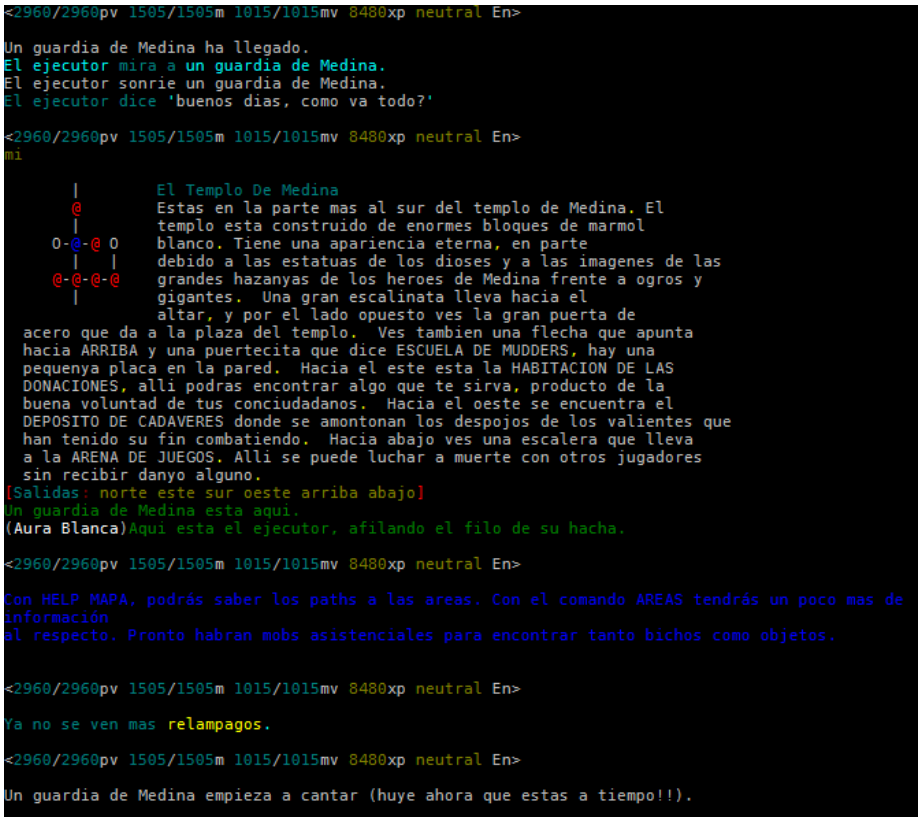


Ilustración 3 – Imagen de un videojuego de tipo MUD

En 1993, id Software lanzó DOOM [6], un popular videojuego de disparos en primera persona (FPS) el cual sería otro hito dentro de los videojuegos multijugador. Una de sus características principales fue el uso de una arquitectura *Peer To Peer* (P2P por sus siglas en inglés), en contraposición al modelo tradicional de cliente-servidor que se ha estado comentando anteriormente. DOOM utilizaba el protocolo IPX (*Internetwork Packet Exchange*) sobre LAN, proporcionando una buena experiencia de juego por aquella época.

En la arquitectura de DOOM los distintos nodos establecían una comunicación por *Broadcast*, donde los paquetes se mandaban a todos los nodos conectados a la red, incluso a los que no eran jugadores. Esto tenía un problema y es que saturaba rápidamente la red. Por ello, en el siguiente título llamado DOOM II, los paquetes de datos eran únicamente transmitidos a aquellos nodos conectados a la red que sí fuesen jugadores.

Ilustración 4 – Imagen del videojuego DOOM de 1993

Otro de los videojuegos que marcó un antes y un después en el multijugador fue Quake [7], un título desarrollado por id Software en 1996 con una arquitectura cliente-servidor. Este título de disparos en primera persona (FPS) introdujo varios conceptos clave como los *Dumb Clients* [8] los cuales eran terminales que únicamente actualizaban el estado del juego en base a los mensajes que recibían por parte del servidor. Todos los cálculos y decisiones eran tomadas por el servidor y los clientes únicamente debían de aplicar dichos cambios en la copia local del mundo del juego que poseían. Como el servidor era autoritativo y los clientes eran *Dumb Clients*, todos ellos permanecían sincronizados durante toda la partida. Esto hizo que apareciese el problema de la latencia debido a que los mensajes del servidor tardaban mucho tiempo en llegar. Por ello, en diciembre de 1996 llegó un hito muy importante para Quake ya que se lanzó QuakeWorld [9] una actualización donde John Carmack, uno de los desarrolladores, explicaría en uno de sus *logs*, los cuales eran registros cronológicos de los distintos cambios llevados a cabo en Quake [10] que esta traería mejoras en la calidad del juego en línea, entre ellas la implementación de la predicción en el lado del cliente. Esto mejoraría la responsividad del juego frente a las acciones del jugador ya que los clientes ya no necesitarían esperar a los mensajes de actualización por parte del servidor, sino que ellos predecirían la reacción de los *inputs* (conjunto de teclas y/o botones presionados en un instante de tiempo determinado) del jugador y, cuando llegasen los mensajes del servidor, se comprobaría en caso de tener que realizar alguna corrección.

A pesar de las grandes mejoras que trajo QuakeWorld, este no fue el primer título en introducir la técnica de la predicción en el lado del cliente al mundo de los videojuegos. En enero de 1996 se lanzó Duke Nukem 3D [11] el cual ya traía desde los inicios de su lanzamiento esta técnica implementada. Pero durante mucho tiempo este hito no se le reconoció a él, sino al Quake. Debido a esta incertidumbre Ken Silverman, creador del motor de videojuegos *Build Engine* [12] sobre el que se había desarrollado Duke Nukem 3D, se pronunció y comentó en una entrevista [13] que su motor había tenido implementado la predicción en el lado del cliente desde el primer día del lanzamiento de Duke Nukem 3D.

Ilustración 5 – Imagen del videojuego Quake de 1996

3.2 Casos de estudio

Arquitectura de Starsiege Tribes

Otro videojuego que, al igual que Quake, posee una arquitectura cliente servidor, pertenece al género de disparos en primera persona y merece ser mencionado en este apartado debido a sus innovadoras técnicas es el Starsiege Tribes, lanzado a finales de 1998 y cuyo modo multijugador era capaz de soportar hasta 128 jugadores conectados a una misma partida. Esto por aquel entonces era todo un logro, pero para conseguir estas cifras tuvieron que tomar ciertas consideraciones en la etapa de desarrollo [14]. En primer lugar decidieron utilizar el *User Datagram Protocol* (UDP, por sus siglas en inglés), un protocolo de red no fiable, para el envío de paquetes entre clientes y servidor. Esto traía consigo ciertos problemas cuando se quería asegurar que la información que se enviase llegase con éxito al destino. Por lo que para gestionar estos problemas, los desarrolladores de Tribes definieron 4 categorías de datos que iban a ser transmitidos por la red:

Datos no garantizados. Estos son todos aquellos datos que no se consideran esenciales para el juego y por lo tanto no serán retransmitidos en caso de que se pierdan.

Datos garantizados. Estos, a diferencia de los anteriores, si son importantes de cara al juego y deberán de ser retransmitidos en caso de que no lleguen al destino.

Datos acerca del estado más reciente. Este grupo engloba aquellos datos donde solo importa la versión más actual.

Datos garantizados urgentes. Estos son aquellos datos que deben de llegar al destino en la mayor brevedad posible.

Otra decisión importante que decidió tomar el equipo de desarrollo fue usar una arquitectura cliente servidor, en vez de P2P. Esto significa que, los clientes ya no se conectan todos con

todos sino que únicamente necesitan establecer conexión con un servidor central. Esta decisión fue en parte una de las que pudo hacer que cada partida soportase 128 jugadores [15] debido a que, en una red P2P cada cliente, el cual está conectado con el resto de clientes, necesitaría un ancho de banda de O(N). Por lo que si hay N clientes conectados, el ancho de banda total de la red P2P ascendería a una magnitud cuadrática de O(N^2). En cambio, usando una arquitectura cliente servidor cada cliente sólo tendría que comunicarse con el servidor por lo que el ancho de banda quedaría constante. Únicamente el servidor, el cual si que necesita conectarse con el resto de clientes, tendría un ancho de banda de O(N).

Además, Tribes posee una implementación de su arquitectura de red dividida en 3 capas como bien se muestra en la Ilustración 6 [14]

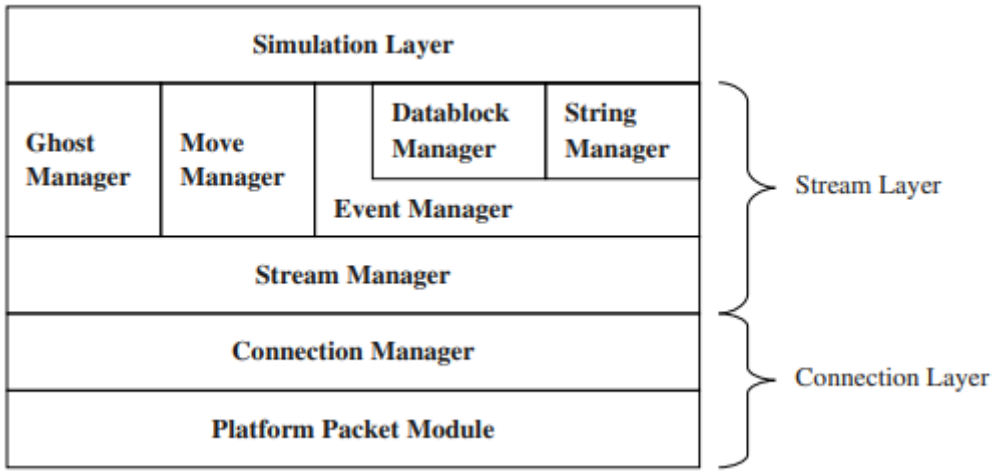


Ilustración 6 - Componentes y capas de la implementación de Starsiege Tribes

Se va a empezar describiendo la capa de más bajo nivel llamada `Connection Layer`. Esta capa se encarga de enviar los paquetes del cliente al servidor y/o viceversa. Posee dos componentes. En primer lugar, el `Platform Packet Module` es el encargado de mandar los distintos tipos de datos descritos anteriormente por la red atendiendo a sus distintas características. Este es el único componente el cual es específico de cada plataforma.

El otro componente de la `Connection Layer` es el `Connection Manager` el cual es el encargado de abstraer la conexión entre dos máquinas del resto de componentes en la capa superior. Cuando recibe datos del `Platform Packet Module` este los redirige hasta el `Stream Manager` y viceversa. Además, como todo se sustenta sobre un protocolo de comunicaciones no fiable no se puede garantizar el envío de los paquetes. Es por ello por lo que se implementó un sistema de notificaciones el cual informa a las capas superiores sobre el estado de la entrega de cada uno de esos paquetes. Por ejemplo, como comentan los desarrolladores, tanto si un paquete se pierde o llega desordenado se marcará como perdido. Por otra parte, este componente únicamente se encarga de notificar del estado de entrega de un paquete, pero todas las tareas de gestión de las reacciones de estos estados de entrega es

tarea de las capas superiores (p.ej. si se notifica un paquete como perdido habrá que valorar si reenviarlo). En la Ilustración 7 se puede apreciar una representación gráfica de este sistema de notificaciones entre los componentes `Connection Manager` y `Stream Manager` [14].

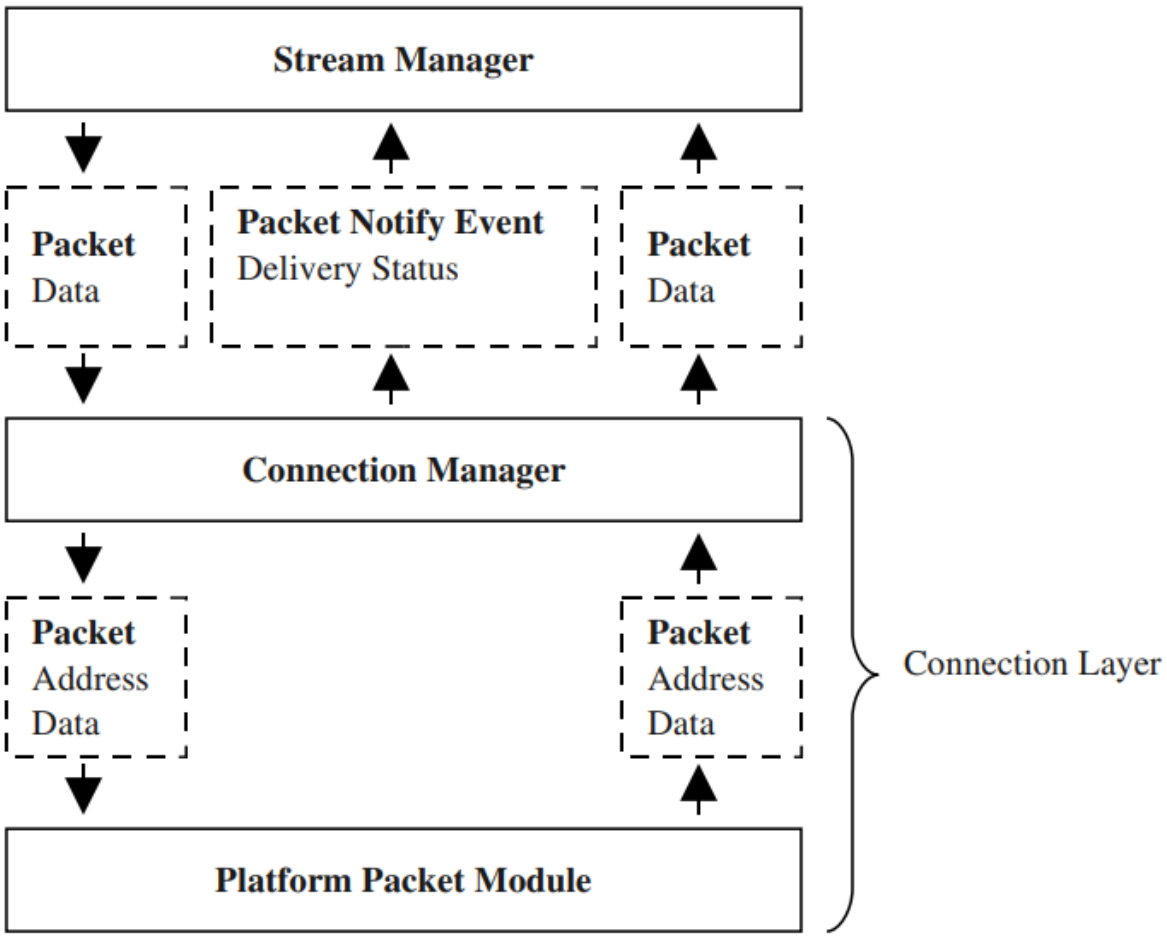


Ilustración 7 - Sistema de notificación del estado de entrega de paquetes en Starsiege Tribes

La segunda capa de la implementación es la `Stream Layer`, cuyo componente principal llamado `Stream Manager` será el encargado de gestionar distintos parámetros del envío de datos tales como el tamaño de los paquetes o la tasa de envío. Estos parámetros irán variando según sean las condiciones de red. Además, otra tarea del `Stream Manager` será priorizar aquellos datos que sean más importantes que otros, por lo que si se alcanza el tamaño máximo de un paquete, la información restante deberá de esperar al siguiente envío de paquete originando una actualización del estado parcial o *Partial Update*. El resto de componentes en esta capa deberán de proporcionar los datos específicos de cada componente al `Stream Manager` y al recibir las notificaciones del estado de entrega de los paquetes gestionar sus reacciones en base al tipo de datos que manejen. A continuación, se detallarán los aspectos más importantes de cada uno de ellos.

El `Ghost Manager` será el encargado de crear copias (llamadas *ghost*) de los objetos que persistan en el juego. Para poder ser capaz de soportar hasta 128 jugadores en la misma partida, el sistema de *ghosting* únicamente replicará y enviará información de aquellos

objetos que son relevantes para cada cliente en un instante determinado. Si el objeto pasa a ser relevante se enviará la información de estado de ese objeto y se creará un *ghost*. En cambio, si dicho objeto sale fuera de la zona de relevancia del cliente el *ghost* será eliminado. De esta forma, se consigue reducir el número de datos enviados por la red. Además, este sistema también soporta objetos que siempre sean relevantes, los cuales estarán marcados como *ghost always* [14].

El `Move Manager` será el encargado de enviar al servidor los *inputs* relacionados con el movimiento en la mayor brevedad posible. Debido a esto, los datos enviados por este componente entrarán dentro de la categoría de datos garantizados urgentes y poseerán la máxima prioridad para el `Stream Manager` a la hora de decidir qué datos enviar en el próximo paquete [14]. Si estos datos no se encontraran en esta categoría podría resultar frustrante para algunos jugadores por el hecho de que los jugadores podrían estar disparado a una posición no actualizada de un jugador debido a que la nueva información todavía no ha podido ser mandada o no ha llegado todavía [15].

El `Event Manager` se encarga de gestionar todos los eventos en el juego tales como cuando un arma ha sido disparada. Estos eventos se comportan como una especie de Llamadas a procedimiento remoto (RPC, por sus siglas en inglés) las cuales permiten ejecutar código en remoto. Estos eventos poseen distintas prioridades y pueden formar parte tanto de la categoría de datos garantizados como de datos no garantizados. Estos eventos se van almacenando en una cola y se van sacando hasta que el paquete esté lleno, la cola esté vacía o la ventana de eventos esté llena. Además, si el `Stream Manager` le notifica que un evento de la categoría de datos garantizados se ha perdido, el `Event Manager` lo volverá a introducir en la cola de eventos para su retransmisión.

Por último, está la capa de más alto nivel llamada `Simulation Layer` [14] cuyas tareas más importantes serán avanzar la simulación en intervalos de tiempo fijos, determinar qué objetos pasan a ser o dejan de ser relevantes para cada cliente y de ejecutar una predicción en el lado del cliente.

Arquitectura de Quake III

Un año más tarde, concretamente en diciembre de 1999, se lanzó el **Quake III Arena**, cuya arquitectura cliente servidor distaba en parte de la de **Starsiege Tribes**. A diferencia del título de **Tribes**, **Quake III Arena** no realizaba actualizaciones parciales del estado de juego [16], sino que usaba un sistema de *snapshots*, estructuras donde se empaquetaba la totalidad del estado de juego ocurrido en un instante de tiempo determinado. Como consecuencia, esto haría que el tamaño de los paquetes incrementase, por ello los desarrolladores introdujeron una serie de optimizaciones para contrarrestarlo. En primer lugar, en vez de transmitir la totalidad del *snapshot*, decidieron únicamente transmitir aquellos aspectos que habían cambiado respecto al último estado de juego. A esto se le conocía como *delta-snapshots*. Por otra parte usaron técnicas de compresión de datos como el algoritmo de Huffman coding [17] el cual reducía aún más el tamaño de los paquetes.

La arquitectura de este título posee varios aspectos clave. Por una parte, el código del cliente se encuentra separado en archivos diferentes del código del servidor [16]. Únicamente existen

una serie de ficheros los cuales son compartidos por ambas instancias del juego, como por ejemplo el módulo de comunicaciones `qcommon` donde se gestionaba todo lo relacionado con la transmisión de paquetes a través de la red. Este aspecto hacía que extender el juego con nuevas funcionalidades fuese más difícil. Además, si se quería usar el *Engine* de Quake III Arena para implementar una versión de un solo jugador, también presentaba un alto grado de dificultad [8]. Debido a esto, los desarrolladores lanzaron dos ejecutables distintos, uno dedicado al multijugador el cual haría de cliente y otro que proporcionaría la experiencia de un solo jugador. Otro punto interesante eran los distintos estados de juego los cuales tienen que permanecer sincronizados. El servidor, posee toda la información del estado de juego global, llamada `Game` en la Ilustración 8, mientras que el cliente, únicamente posee un estado de juego local formado por aquellos objetos relevantes para el cliente, llamada `cGame` en la Ilustración 8. Por otra parte, el cliente implementaba un sistema tanto de interpolación como de extrapolación usando las dos últimas actualizaciones del servidor para hacer que las transiciones entre los distintos estados de juego que llegaban del servidor se vieran más fluidos [8]. También, posee otros sistemas encargados de renderizar el estado local de juego por pantalla, la interfaz de usuario… [16]

Como se comenta en el paper oficial [16], la representación visual hecha en la Ilustración 8 es únicamente representativa para consolidar una idea de cómo los distintos componentes interactúan entre sí, ya que a la hora de la implementación [18] se rompen algunas de esas barreras.

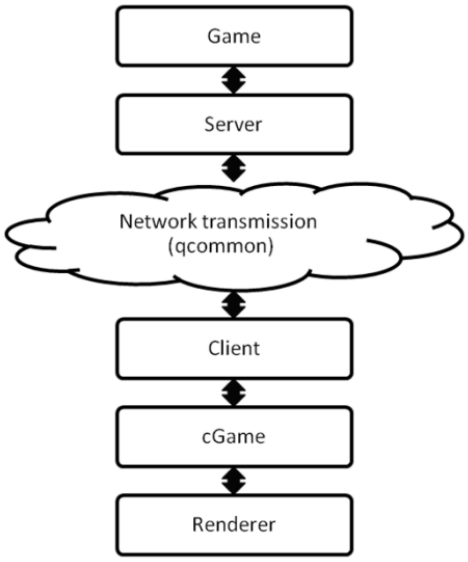


Ilustración 8- Arquitectura del Quake III Arena

A continuación, se pasará a hablar de cómo se realiza la comunicación entre el cliente y el servidor. En esta arquitectura, el cliente únicamente envía la información de las acciones que el jugador ha tomado en cada fotograma de la partida (p.ej. información del movimiento o del disparo de las armas). Esta se guarda en una estructura llamada `usercmd_t` y se envía al servidor. El servidor, de forma periódica, avanza la simulación con la información de los

comandos recibidos por parte de los clientes. Esto se hace a través del módulo PMOVE, representado en la Ilustración 9, el cual a partir del estado actual de cada jugador y los nuevos comandos recibidos por parte del cliente calcula el nuevo estado de dicho jugador. Este estado se almacena en una estructura de tipo `playerstate_t` la cual contiene información como su posición, velocidad de movimiento, entre otros.

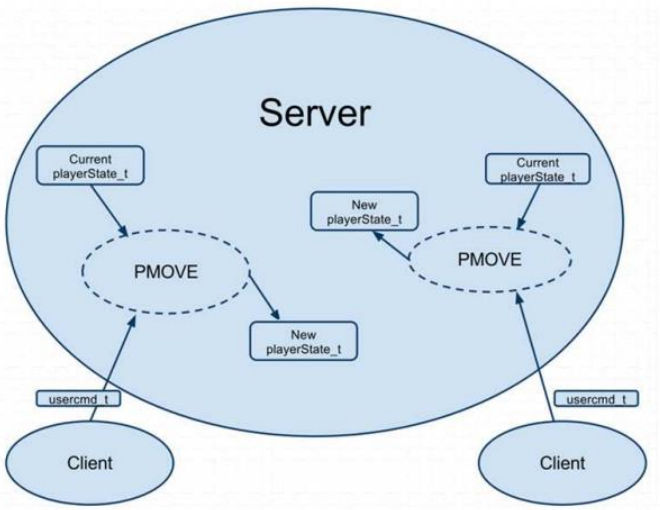


Ilustración 9 - Representación del funcionamiento del módulo PMOVE en Quake III Arena

Una vez se ha avanzado la simulación y se ha actualizado el estado de juego global, el servidor comienza a crear y enviar las actualizaciones al resto de clientes conectados. Para ello, por cada cliente se deberá de generar una *snapshot* únicamente con aquella información relevante para ellos. Esta idea ya se aplicó en Starsiege Tribes para optimizar la cantidad de información que era necesaria enviar a cada cliente. En el caso de Quake, este algoritmo consiste en hacer una criba y descartar todas aquellas entidades que no son visibles desde la cámara del jugador [19]. De esta manera, solo se incluirán en la *snapshot* aquellas que el cliente pueda ver. Una vez se hayan encontrado todas las entidades visibles, se añadirán a la *snapshot* así como el nuevo estado del jugador controlado por dicho cliente. Estas entidades se almacenan en una estructura llamada `entityState_t`, la cual es una versión simplificada de `playerState_t`, y pueden ser jugadores enemigos, balas, puertas, etc. Una de las razones por las que se separa al jugador controlado por un cliente, representado por la información de `playerState_t`, del resto de jugadores enemigos, representados por la información en `entityState_t`, es por una cuestión de optimización. La información de `playerState_t` es mucho mayor que la almacenada en `entityState_t`. Además, mucha de esa información es innecesaria para representar a los jugadores enemigos. Como comenta Jay en uno de sus artículos [20], se debería poder renderizar a los jugadores controlados por el resto de clientes usando únicamente información relacionada con la posición, rotación, animación y una indicación de tiempo por parte del servidor. Por ello, existe mucha información en `playerState_t` como por ejemplo el estado del arma que únicamente es relevante para el cliente que controle a ese jugador. Una vez creada la *snapshot* personalizada para cada cliente, se le aplicarán las dos compresiones citadas anteriormente de delta para únicamente

incluir los cambios respecto al último estado de juego y de Huffman coding para reducir aún más el tamaño de los datos. Finalmente se empaquetará en la estructura `msg_t`, la cual es la que contiene toda la información adaptada para los sistemas de bajo nivel y lista para ser transmitida.

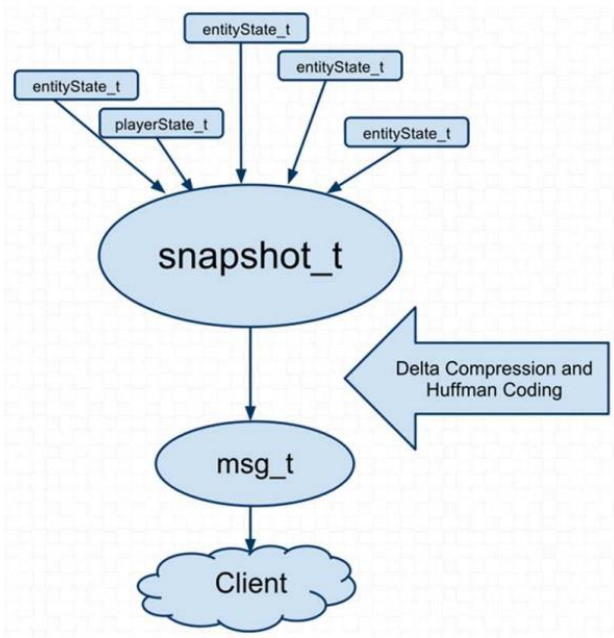


Ilustración 10 - Generación de una snapshot en Quake III Arena

Una vez esta *snapshot* llegue al cliente, este deberá de deshacer las técnicas de compresión aplicadas en el servidor para obtener la *snapshot* resultante. Especialmente para convertir una *delta-snapshot* en una *snapshot* completa, el cliente deberá de tener un *buffer* de *snapshots* en el que se almacene el historial de snapshots recibidas más reciente para llevar a cabo esta conversión. Esto es debido a que, como comenta Glenn Fiedler en uno de sus artículos [21], las *delta-snapshots* muestran únicamente los cambios sucedidos a partir de una *snapshot* anterior. Es por ello por lo que, si la *snapshot* 80 viene codificada a partir de los cambios sucedidos desde la *snapshot* 70, el cliente deberá de conocer cúal es la *snapshot* 70 para así poder decodificar la 80. Cuando se haya obtenido la *snapshot* final, el cliente procesará los comandos, el estado del jugador y las entidades recibidas por parte del servidor.

Arquitectura de DOOM III

Uno de los videojuegos que marcó un hito en la historia de los videojuegos multijugador fue DOOM, lanzado en 1994 fue uno de los primeros títulos en 3D de disparos en primera persona. Este estaba construido sobre una arquitectura P2P, lo que significaba que, a diferencia de los títulos anteriores donde cada cliente únicamente establecía conexión con un servidor, en el DOOM original cada cliente se conecta con el resto de clientes de la partida. En consecuencia, ahora los comandos que introduzca un cliente deben de ser enviados al resto de clientes de

la partida. Estos paquetes de comandos almacenaban información acerca del movimiento y ciertas acciones como disparar o usar. Por ello, al igual que un cliente enviaba sus comandos al resto, también tenía que recibir los comandos del resto. Estos últimos se guardaban en un *buffer*. Cuando hubiesen llegado todos los comandos, el estado de juego avanzaría de forma totalmente independiente a la recogida de comandos.

Este sistema presentaba varios problemas [8]. El primero de todos era que existía un retardo entre que el jugador introducía el comando y lo veía reflejado en el juego. Esto era causado debido a que antes de avanzar el estado de juego, cada cliente debía de conocer los comandos de cada uno del resto de clientes, por lo que se generaría un retraso que dependería del jugador con la conexión más lenta. Si la conexión se realizase mediante una red local (LAN) este problema apenas sería perceptible ya que, como comentan los desarrolladores, este *ping* estaría por debajo de los 10 milisegundos. En cambio, si la conexión fuese a través de internet este *ping* podría superar fácilmente los 100 milisegundos creando así un retardo perceptible por el jugador.

Otro problema de esta arquitectura era que, a lo largo del tiempo, se podrían generar inconsistencias entre los estados del juego en los distintos clientes (p.ej. errores en operaciones de coma flotante que se van acumulando). Estas inconsistencias dependían muchas veces por el tipo de hardware. Por ejemplo, en el paper oficial [8] los desarrolladores comentan que si se introducían objetos que se moviesen en base al volumen de otros elementos del entorno, en caso de que dos clientes tuviesen tarjetas de audio distintas con tasas de muestreo diferentes debido a la velocidad de los ordenadores se generarían inconsistencias. Por este mismo problema, el juego no podría ser multiplataforma.

Por otra parte, existía otro aspecto negativo que presentaba esta solución P2P y estaba relacionado con el ancho de banda disponible. El número de información que cada cliente debía de transmitir a través de la red era directamente proporcional al número de jugadores conectados a la partida. Por lo que si por ejemplo, cada paquete de comandos tenía un tamaño de 10 bytes, para N clientes conectados cada uno de ellos tendría que mandar (N – 1) * 10 bytes de información por cada *tick*, es decir, por cada intervalo de tiempo en el que el cliente realiza actualizaciones y procesa eventos, ya que se deberá de enviar los comandos al resto de clientes. Considerando el ancho de banda que ofrecían los módems de aquella época este aspecto se convertía en una barrera a nada que el número de clientes aumentase.

Otra limitación que tenían los jugadores era la de no poderse unir a una partida que ya había comenzado. Debido a que únicamente se transmitían los comandos de cada cliente, el estado de juego únicamente podía replicarse si todos ellos habían empezado a simular en el mismo instante de tiempo. Pero si alguien intentaba replicarlo en un punto intermedio de la partida sería prácticamente imposible por el hecho de no disponer del estado de juego en el momento en el que se conectó.

Como último problema el estado de juego era fácilmente hackeable. Como comentan los desarrolladores, debido a que cada cliente poseía la totalidad del estado de juego, el cliente podría manipular esta información para hacer visibles objetos que se supone que no debería de ver como por ejemplo al resto de jugadores.

Como se comenta en el paper [8], una solución sería usar un *Packet Server* el cual es una arquitectura tanto con elementos P2P como cliente servidor. En esta arquitectura, en vez de que cada jugador establezca una conexión con el resto de jugadores, solo se establece conexión con un jugador. Este jugador, aparte de hacer de cliente, también hará de *servidor de relay.* Esto significa que cada cliente enviará sus comandos al jugador que sea servidor y esté los redigirá al resto de clientes. Con esta arquitectura, se conseguía reducir el retardo de los jugadores con buena conexión, ya que si el paquete de un cliente tardaba mucho en llegar, el *packet server* enviaría el último paquete que tuviese de nuevo. De esta forma solo el cliente con mala conexión experimentaría los efectos negativos de la misma. Por otra parte, esta arquitectura presentaba un problema ya que si el cliente que tuviera el *packet server* se desconectaba, la partida no podría continuar.

Otra solución que se comenta, la cual es más adecuada para videojuegos de disparos en primera persona de ritmo rápido, es la arquitectura cliente servidor. Esta arquitectura, la cual ya ha sido comentada en juegos como Starsiege Tribes o Quake III Arena mitigaría muchos de los problemas presentados en la arquitectura P2P del DOOM original. Por una parte, acabaría con el problema de la desincronización de los estados de juego de los distintos clientes ya que, al igual que se explicó en Quake III Arena, en esta arquitectura el servidor enviaría *snapshots* al resto de clientes conteniendo el estado del jugador así como el resto de entidades relevantes para dicho jugador en un instante de tiempo determinado, por lo que la recreación de este estado siempre seguiría de forma fiel a la versión que tuviese el servidor. Además, como cada cliente únicamente tendría una versión local del estado de juego del servidor, hackear el juego haciendo uso de trucos para ver al resto de jugadores sería mucho más difícil puesto que, al igual que el Quake III Arena, únicamente se enviarían aquellas entidades que fuese visibles desde la cámara del jugador. Otro problema del DOOM original con el que se acabaría sería la imposibilidad de que nuevos jugadores puedan unirse a en mitad de una partida. Ahora, como el servidor estaría mandando de forma regular *snapshots* al resto de clientes, sería muy fácil recrear dicho estado de juego a partir de la *snapshot* que llegase y comenzar a jugar.

Aunque en esta arquitectura cliente servidor existen algunos problemas mencionados en la arquitectura P2P del DOOM original que a priori no se solucionarían. Este es el caso del problema del *ping*. El jugador, tendría que esperar a que los comandos se enviasen desde cliente hasta servidor, este los procesara, y enviase la respuesta de vuelta al cliente para poder renderizarlo por pantalla. Esto se podría corregir aplicando la técnica de la predicción en el lado del cliente la cual fue mencionada anteriormente con la actualización de QuakeWorld y que se ha usado en futuros títulos de la saga. Esta técnica aumentaría la responsividad que el jugador sentiría en el juego minimizando así el problema del *ping*. Aunque, como comentan los desarrolladores de Valve [22], si se dan algunos errores a la hora de predecir en el cliente, cuando lleguen los resultados oficiales del servidor habrá que realizar una corrección creando un cambio perceptible.

Por otra parte, este problema del *ping* afectaría no solo afectaría al movimiento del personaje sino también al sistema de disparos. Esto es debido a que cuando un jugador presiona la tecla encargada de disparar, el cliente debe primero mandarla al servidor. El problema ocurre al llegar al servidor ya que el jugador enemigo puede haberse movido y no encontrarse ya en la

posición en la que lo vió el atacante dando como resultado un disparo fallido. Para paliar esto se crearon las técnicas de compensación de *ping*. Como comenta el desarrollador de Valve Yahn Bernier [22], para minimizar este efecto, el servidor almacena un historial de posiciones de todos los jugadores. Cuando llega el comando de disparo de algún jugador, el servidor calcula la latencia con dicho cliente y busca en el historial de posiciones aquella que se encuentre antes de que el jugador haya disparado. Una vez encontrada, se rebobina el estado de juego hasta ese instante y se simula el disparo. De esta forma el servidor podrá recrear el estado que vio el cliente a la hora de disparar. Tras esto, se deshace el rebobinado. Aunque esta solución trae claros beneficios, también posee algunas inconsistencias [8], puesto que aquellos jugadores con buena conexión podrían experimentar efectos como ser alcanzados por un disparo tras ponerse a salvo detrás de una cobertura.

La arquitectura que sigue DOOM III es muy parecida a la de Quake III Arena mencionada anteriormente. En ambas existe un cliente que envía únicamente comandos al servidor y predice los estados futuros del jugador al que controla. Por otra parte, el servidor autoritativo recibe los comandos de los jugadores, los procesa y empaqueta los cambios del estado de juego en una *snapshot* personalizada con la información relevante para cada cliente la cual envía de vuelta a cada uno de ellos.

La arquitectura de DOOM III presenta varias ventajas frente a la de Quake III Arena. Una de ellas estaba relacionada con la flexibilidad. Como se mencionará más adelante, en DOOM III las estructuras de datos encargadas de transmitir el estado de las entidades no serán inmutables por lo que se podrán añadir parámetros adicionales que necesiten ser sincronizados. Por otra parte en esta arquitectura no existe la separación de código entre el cliente y el servidor sino que este es compartido. Esto resuelve el problema que poseía Quake III Arena ya que este código centralizado ofrece una mayor facilidad a la hora de implementar nuevas funcionalidades o un modo de un solo jugador.

En DOOM III el servidor avanza el estado de juego sin esperar a recibir los comandos del resto de clientes. Si algún comando de algún cliente no llega a tiempo, se duplicará y usará el último que se haya recibido. Por ello, cada cliente intenta adelantarse un poco al servidor para que sus comandos lleguen a tiempo incluso si existe alguna fluctuación de la latencia en dichos mensajes. Para llevar esto a cabo, el cliente realiza operaciones de predicción del estado de las entidades. Mientras no se reciban nuevas *snapshots* por parte del servidor, el cliente predice los nuevos estados de estas entidades. Cuando finalmente llega una *snapshot* el cliente sobrescribe el estado predicho por el que traía la *snapshot* y, debido a que dicha *snapshot* pertenece a un estado de al menos la duración de una unidad de ping en el pasado, se rebobina la simulación para volver a predecir a partir del nuevo estado de la *snapshot* con el objetivo de corregir posibles errores de predicción que se hayan generado. En la Ilustración 11 se puede ver un ejemplo de este proceso sacado del paper oficial [8]. A pesar de que esta predicción se realice únicamente con aquellas entidades que son visibles desde el punto de vista del jugador, puede llegar a resultar costoso volver a predecir varios fotogramas consecutivos cada vez que el cliente recibe una nueva *snapshot*. De hecho, cuanto mayor sea la latencia, mayor será el intervalo de tiempo hasta el que haya que rebobinar y, como consecuencia, mayor será el número de fotogramas que habrá que volver a predecir.

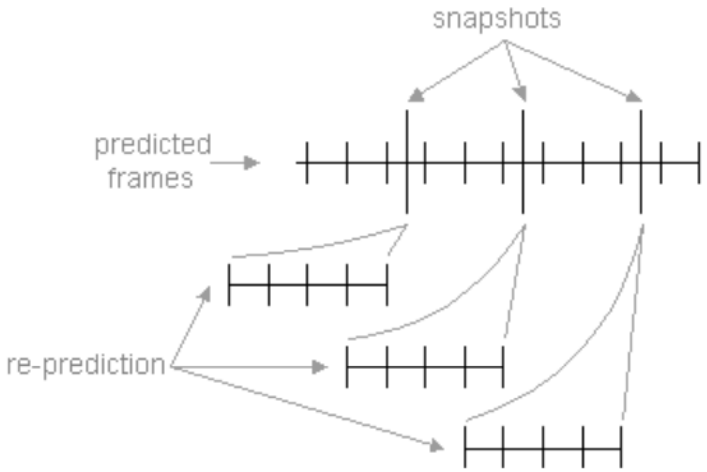


Ilustración 11 - Proceso de predicción en el cliente del DOOM III

Por otra parte, los desarrolladores comentan que en un FPS todo se calcula de forma determinista excepto los comandos que introduce el jugador. Además, en DOOM III la recogida de comandos se realiza 60 veces por segundo, al igual que el número de fotogramas por segundo, pero es muy poco probable que el jugador pulse las teclas a una velocidad de 60 veces por segundo. Por ello, existe una probabilidad de que durante varios fotogramas el comando de un jugador sea el mismo, ofreciendo así la posibilidad de que la predicción de en el cliente de los futuros estados de las entidades a partir del comando actual tenga más probabilidades de éxito. Debido a esto, cada *snapshot* trae el último comando del resto de jugadores para poder llevar a cabo esta predicción. Una ventaja del apartado de predicción del DOOM III frente al Quake III Arena es que a diferencia del Quake, en el DOOM III tanto el servidor como el cliente ejecutan el mismo código del módulo de predicción, incluyendo físicas. El servidor lo usa para avanzar el estado de juego mientras que el cliente para desempeñar la tarea de predicción comentada anteriormente. Gracias a esto, desarrollar un modo de juego de un solo jugador o implementar nuevas funcionalidades no será tan difícil.

Por otra parte, al igual que Quake III Arena, DOOM III utiliza el protocolo UDP para la comunicación entre el servidor y los clientes. Esta se encuentra formada por dos capas adicionales, las cuales pueden apreciarse en la Ilustración 12.

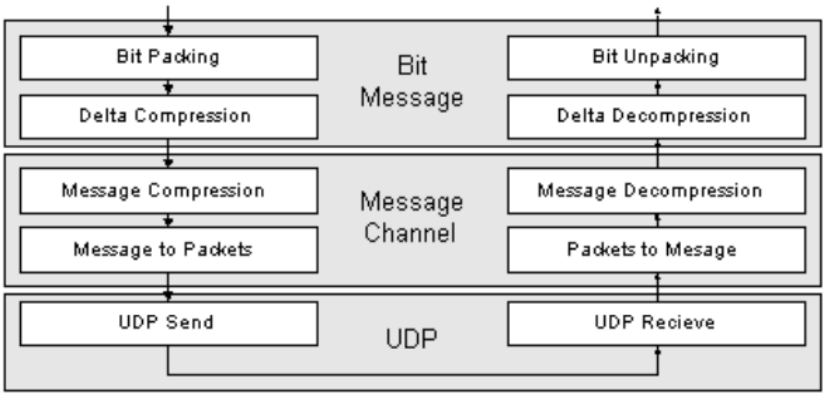


Ilustración 12 - Capas de procesamiento de mensajes en DOOM III

La capa de `Bit Message` [8] es la encargada de aplicar la compresión delta vista en el Quake III Arena además de otro tipo de compresión llamada *Bit Packing* [23] la cual quita aquellos *bits* que no son relevantes y empaqueta lo que queda tras este proceso. Un ejemplo de este algoritmo, como se comenta en el paper oficial [8] podría ser la vida del jugador, la cual se almacena como un entero de 32 *bits* y cuyo rango va desde los 0 puntos de vida hasta los 100. En este rango solo 7 *bits* de los 32 que posee el entero son utilizados, por lo que sabiendo esto se puede únicamente trasmitir esos bits relevantes con la garantía de no perder información. Este algoritmo únicamente ofrece entre un 10 y un 15 por ciento de compresión ya que muchas de las variables almacenadas como posiciones u orientaciones son tipos flotantes de 32 *bits* a los cuales no se les puede aplicar a priori el algoritmo de *Bit Packing*. Una opción sería dividir el exponente y la mantisa del flotante aunque esto supondría redondear la mantisa hasta ciertos decimales y perder algo de precisión en dicho proceso creando así futuros problemas por ejemplo a la hora de la predicción en el cliente.

La capa de `Message Channel` es la capa de más bajo nivel implementada sobre la de UDP la cual provee la posibilidad de mandar tanto mensajes *reliable* como *unreliable*. La mayoría de la información que se transmite en el juego será de tipo *unreliable*, es decir, que no se garantizará que vaya a llegar a su destino. Sin embargo, existe un pequeño porcentaje de los mensajes que son necesarios enviar de forma *reliable*, es decir, garantizando que el destino va a recibirlos, como por ejemplo algunas actualizaciones importantes. La forma en la que se envían los mensajes *reliables* es la siguiente: Estos mensajes se almacenan en un *buffer* y se envían junto con cada mensaje *unreliable*. Cuando llega el *acknoledgement* (ACK por sus siglas en inglés), mensaje que informa de que un paquete de datos ha llegado correctamente, por parte del receptor, se saca dicho mensaje del *buffer* para no incluirlo más en los siguientes mensajes *unreliable*. De esta forma, se garantiza que los mensajes *reliable* siempre llegaran a su destinatario. Normalmente estos mensajes *reliable* suelen tener un tamaño muy pequeño para que no se conviertan en un problema. Otra tarea de la capa del canal de mensajes es notificar a las capas superiores eventos relacionados con la conexión, por ejemplo el momento en el que se ha establecido la conexión. También divide aquellos mensajes de gran tamaño en paquetes más pequeños para evitar la fragmentación de los paquetes y los problemas que puede conllevar [24]. Por último, esta capa se encarga de realizar una última compresión usando el algoritmo de *Run-Length Encoding* (RLE por sus siglas en inglés) [25] el cual es un método de compresión de datos sin pérdida basado en la repetición de valores consecutivos, también llamados *runs* en el algoritmo, dentro de una secuencia de datos. En el caso de DOOM III, usar un tamaño de 3 *bits* para la representación de las *runs* era una solución óptima en la práctica ya que conseguían un ratio de compresión de 4:1, es decir, de un 75% de compresión. Este ratio se puede mejorar agrupando la información de la *snapshot* según la frecuencia con la que esta cambia. Esto es debido a que como la información que no cambia es representada en la *delta-snapshot* con un 0, si se agrupan los valores con una menor frecuencia de cambio se podrían generar secuencias de 0 más largas resultando así en una mayor compresión por parte del algoritmo de RLE.

Por otra parte, dentro de los mensajes *unreliable* estos pueden tener dos cabeceras distintas dependiendo de la dirección en la que se transmitan. Si se transmiten desde el servidor a un

cliente esta cabecera tendrá únicamente dos campos como se puede ver en la Ilustración 13, donde por una parte el servidor enviará un identificador de juego (`game id`) para que los clientes se aseguren de que han cargado el mapa y las configuraciones correctas. Además el servidor enviará de qué tipo es dicho mensaje (`message type`).

32 bits	game id
8 bits	message type

Ilustración 13 - Cabecera de mensaje unreliable por parte del servidor en DOOM III

En el caso de que se transmita desde un cliente hasta el servidor, esta cabecera será algo más larga como se muestra en la Ilustración 14. Por una parte, los campos mencionados anteriormente seguirán estando y se añadirán dos campos nuevos. Uno de ellos será la secuencia de la última *snapshot* recibida por dicho cliente la cual será útil en el servidor para realizar una correcta compresión delta de la próxima *snapshot*. Por último, el cliente enviará también la secuencia del último mensaje recibido por parte del servidor para que el servidor pueda verificar que dicho cliente haya recibido el mensaje con instrucciones para cargar la nueva configuración y mapa del juego. Esto se hace como forma de *acknoledgement*. En este caso, el identificador del juego será usado para que el servidor verifique si el cliente se encuentra en la partida correcta.

32 bits	sequence number of last recieved server message
32 bits	game id
32 bits	sequence number of last recieved snapshot
8 bits	message type

Ilustración 14 - Cabecera de mensaje unreliable por parte del cliente en DOOM III

Al igual que en Quake III Arena, en el DOOM III el elemento principal de la comunicación por parte del servidor es la *snapshot*, un conjunto de información que recrea el estado de juego personalizado para cada cliente en un instante de tiempo determinado. Según el paper oficial el servidor manda *snapshots* a los clientes a una velocidad de entre 10 y 20 Hz. Esta *snapshot*, aparte de poseer la cabecera mencionada anteriormente también posee una serie de campos como se muestra en la Ilustración 15. Por una parte esta posee una serie de campos a los que se le van a aplicar la compresión delta:

1. Los estados de aquellas entidades que son relevantes para el cliente destino, es decir, que se encuentren dentro de su campo de visión. A diferencia de Quake III Arena, en el DOOM III las estructuras de estas entidades no tienen por qué ser inmutables, sino que se pueden añadir variables adicionales que necesiten ser sincronizadas en cualquier momento, aunque esto creará una menor efectividad de la compresión delta.

2. Un *string* el cual contiene aquellas entidades que sí están dentro del campo de visión del cliente.

3. El estado tanto del juego como del personaje que controla dicho cliente.

Por otra parte, en cada *snapshot* también se guardará la información del último comando de todos los cliente que se encuentre en el campo de visión del cliente destinatario. De esta forma se podrá llevar a cabo el proceso de predicción de entidades comentado anteriormente. Además, se incluyen otras variables como el fotograma o el tiempo en el que se generó, el número de comandos duplicados desde la última *snapshot* por el servidor (Esta parte se explicó anteriormente) o con cuánto tiempo de antelación dichos comandos han llegado. Como se comenta en el paper oficial, este valor debería de ser cercano a cero pero nunca negativo, ya que eso significaría q [26]ue ha llegado con retraso. Con este último dato el cliente deberá de poder adaptar su tiempo de predicción.

32 bits	game id
8 bits	message type: snapshot
32 bits	snapshot sequence number
32 bits	game frame number
32 bits	game frame time
8 bits	number of duplicated user commands
16 bits	client ahead time
variable # bits	delta compressed entity states
variable # bits	delta compressed PVS bit string
variable # bits	delta compressed game & player state
variable # bits	latest user commands from the other clients in the PVS

Ilustración 15 – Información transmitida a través de la red de una snapshot en DOOM III

Por otra parte, el elemento principal de la comunicación por parte del cliente son los comandos del jugador, mostrado en la Ilustración 16. Estos son enviados con una frecuencia 3 o 4 veces mayor que las *snapshots* y su tamaño es mucho menor. En cada uno de estos mensajes existe un campo al que se le aplica la compresión delta, el cual es una secuencia con los distintos comandos que el jugador ha presionado, cada uno de estos comandos es parte de un fotograma. Por ello, al aplicar la compresión delta de esta secuencia, cada comando se comprime en base al anterior. Por otra parte, en estos mensajes se guardará tanto el fotograma en el que se pulsó el primer comando de la secuencia anterior así como el número de comandos que esta contiene.

32 bits	sequence number of last recieved server message
32 bits	game id
32 bits	sequence number of last recieved snapshot
8 bits	message type: user command
16 bits	client prediction in milliseconds
32 bits	game frame number
8 bits	number of user commands
variable # bits	delta compressed user commands

Ilustración 16 – Información transmitida a través de la red de los comandos del jugador en DOOM III

Arquitectura de Age Of Empires

Otro título que ha quedado enmarcado en la historia de los videojuegos multijugador debido a sus aportes fue el Age Of Empires. Este usaba una arquitectura P2P. Mientras se desarrollaba la primera entrega de esta saga, los programadores se toparon con una limitación en la red [27], y es que con la gran cantidad de unidades y objetos interactuables que iba a haber en pantalla, pasar toda la información del estado de cada uno de ellos al resto de clientes sería prácticamente imposible. Una opción hubiese sido limitar el número de unidades con las que el jugador pudiese interactuar en cada turno a un número muy bajo, pero el juego estaba pensado para que el jugador pudiese arrasar ciudades enteras con todo tipo de ejércitos, por lo que establecer un límite de unidades muy pequeño acabaría con esta sensación de guerra a gran escala.

En el Age Of Empires original [15], el número de unidades por cada jugador era de 50. Además, por cada partida en línea el número máximo de jugadores era de 8. Por lo que en el peor escenario el número máximo de unidades que podía haber simultáneamente era de 400 unidades. Podría haberse pensado alguna solución como la implementada en el Starsiege Tribes donde únicamente se mandaría la información del estado de aquellas unidades relevantes para cada cliente. Pero si en la partida se diese un enfrentamiento que involucrase a las 50 unidades de los 8 distintos ejércitos, la gran cantidad de unidades existentes haría inviable enviar toda esa información a través de la red.

Es aquí cuando nació el *Deterministic Lockstep*, una técnica cuyo objetivo no era sincronizar todas las unidades existentes en la partida sino sincronizar únicamente los comandos de los jugadores al principio de cada turno. Si se conseguía recrear la misma simulación de los distintos turnos en cada cliente, únicamente a partir de los comandos introducidos por los jugadores, no sería necesario enviar el estado de cada unidad en el mapa. Esto daba solución a una parte del problema, pero para que todas estas simulaciones den los mismos resultados todos los jugadores tenían que haber recibido todos los comandos del resto de jugadores antes de comenzar la simulación del turno. En caso contrario surgirían discrepancias. De esta forma, se creó un sistema de temporización de turnos donde cada turno duraba 200 milisegundos. Durante este tiempo, cada cliente guardaba los comandos introducidos por el jugador en un *buffer*, el cual se enviaría al final del turno al resto de clientes conectados para llevar a cabo el cálculo de la simulación del nuevo turno. Pero este cálculo no podía comenzar

hasta que todos los clientes hubiesen recibido los comandos del resto de clientes. Por ello, para garantizar de que cada cliente los había recibido y había notificado de vuelta al resto se añadió un retardo en el procesamiento de los comandos de 2 turnos de duración. De esta forma, si los comandos se enviaban en el turno 30 no era hasta el turno 32 que se ejecutaban. Esto creaba un total de 600ms de retardo desde que el jugador introducía el comando hasta que realmente se veía reflejado en el juego.

Además, se tuvo en cuenta el hecho de que cada cliente podría tener tanto una velocidad de procesamiento como unas condiciones de red distintas al resto. Como consecuencia, se creó el `Speed Control`, un sistema encargado de adaptar de forma dinámica la duración de los turnos para que el juego se viese fluido a pesar de las condiciones cambiantes de la red o de las velocidades de procesamiento de las distintas máquinas. Según explica Bettner y Terrano en su paper [27], para conseguirlo, cada cliente tendría que medir la tasa de fotogramas media a la que corría el juego y enviarlo junto con los comandos al acabar cada turno. Además de forma periódica debería medir el *Round-Trip Time* (RTT por sus siglas en ingles), el cual era el tiempo que tardaba un paquete de datos en llegar a su destino y luego regresar al punto de origen, con el resto de clientes y enviar aquel que fuese más alto. Tras esto, el host encargado de analizar estos parámetros calculaba un ajuste ideal tanto para la tasa de fotogramas como para la latencia en base al cliente con las peores condiciones. Tras este proceso, el *host* enviaba los resultados de vuelta al resto de clientes para que estos pudiesen adaptarse.

Tras este desarrollo, el equipo técnico sacó una serie de lecciones aprendidas a la hora de desarrollar un videojuego multijugador [27]:

1. Cada género es diferente, por lo que entender sus mecánicas, retos y limitaciones será clave a la hora de tratar temas como los rangos de latencia permitidos o la optimización necesaria. Por ejemplo, en **Age Of Empires** los jugadores se habían acostumbrado a que sus acciones, debido a la latencia, tardaran aproximadamente unos 500 milisegundos en ser aplicadas al estado de la partida, por lo que era mejor mantener estos valores de latencia que el hecho de que fuesen variando (p.ej. en un rango de entre 80 a 500 milisegundos).

2. Estar constantemente testeando las distintas funcionalidades en red ante las distintas condiciones de la red hará que el equipo pueda detectar qué tareas están generando posibles cuellos de botella o condiciones de carrera para así poder mitigarlos cuanto antes. Además, crear un sistema de métricas fácil de entender para facilitar el trabajo a los distintos testers es otro punto clave para detectar fallos.

3. Programar soluciones multijugador implica un pequeño cambio de mentalidad ya que al no poder fiarse de la red, es necesario que todo sea validado. Puede darse el caso de que un mensaje o incluso varios consecutivos se pierdan, o que tarden más en llegar. Por ello es necesario que el videojuego sepa gestionar todo este tipo anomalías de la red. Por otra parte, para evitar errores de sincronización en el cálculo de la simulación, es necesario que esta no dependa de factores locales como por ejemplo valores aleatorios.

Arquitectura de For Honor

Aparte de Age Of Empires, existieron más títulos los cuales decidieron usar la arquitectura P2P. Uno de ellos fue For Honor. Los objetivos que tenía el equipo de desarrollo eran el de mantener un bajo uso del ancho de banda así como un juego justo.

Al igual que en el Age Of Empires, cada *peer* de For Honor únicamente envía los comandos al resto de *peers* y posteriormente realiza una simulación determinista para obtener estados de juego idénticos. Por otra parte, existe un *buffer* el cual almacena los estados de juego generados tras la simulación durante los últimos 5 segundos. Esto se debe a que, a diferencia de Age Of Empires, los *peers* no esperan a haber recibido los comandos del resto de *peers* antes de comenzar la simulación. Como consecuencia, si llega un comando que pertenecía al pasado, se recupera del *buffer* el estado de juego más cercano a dicho comando en el pasado y se vuelve a simular hasta el instante actual. Según se comenta en la ponencia impartida por una de las desarrolladoras [26], estos procesos de retroceso ocurren en todos los fotogramas aunque apenas son perceptible para el jugador.

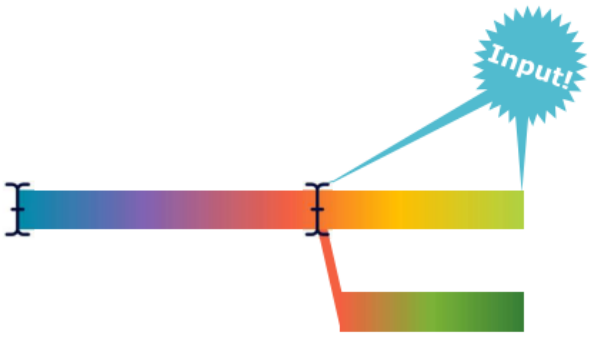


Ilustración 17 - Proceso de resimulación en For Honor

Uno de los grandes problemas de las arquitecturas *Peer To Peer* son las desincronizaciones de los distintos *peers* debido a discrepancias en las distintas ejecuciones de las simulaciones. Como se comenta en la ponencia, hacer que la totalidad del proceso de simulación sea determinista es un gran reto ya que es necesario controlar aquellos elementos susceptibles de divergir como por ejemplo:

1. Las operaciones de coma flotante.

2. Los generadores de números aleatorios, ya que se debe de garantizar en cada *peer* que el generador se inicializa con la misma semilla y que se generan el mismo número de valores aleatorios para no producir una desincronización en la secuencia.

3. El tiempo. En For Honor existían dos tipos de tiempo: Un tiempo que representar la duración que el jugador lleva jugando, el cual únicamente avanzaba y el tiempo simulado el cual es fijo, se actualizaba 30 veces por segundo y es capaz de volver atrás al para volver a simular.

4. El motor de físicas. Para este proyecto se usó el motor de físicas Havok [28] el cual ofrece simulaciones deterministas y *stateful*. El término *stateful* hace referencia a que Havok almacena información de los estados para poder usarla posteriormente en el cálculo de los estados futuros. Esto, como se comenta en la ponencia, traía un problema ya que, debido a la propiedad *stateful* de Havok, a la hora de retroceder el estado de juego para volver a simular se rompía el determinismo dando resultados diferentes.

5. Los tiempos dinámicos de carga. Cada *peer* puede tener una máquina con un *hardware* distinto, por lo que en algunas el juego cargará más rápido que en otras.

Como se puede ver en la lista anterior, hay muchos factores que pueden hacer que nuestra simulación deje de ser determinista. Pero como comenta Jennifer, los desarrolladores pueden dejar algunos aspectos del juego fuera de este determinismo. En For Honor el sistema de animaciones no es determinista ya que existe un pequeño *offset* mostrado en la Ilustración 18 entre la posición simulada, la cual representa donde realmente se encontraba dicho objeto, y la posición de renderizado, la cual muestra dónde está la animación. Si este offset supera un umbral se corrige con un pequeño teletransporte de la posición de renderizado. Aparte de las animaciones otros elementos no deterministas son los efectos visuales o los sonidos. Esto supone una ventaja ya que si por ejemplo el personaje controlado por el jugador ataca con su espada a otro jugador resultando en un ataque exitoso, se instancia un efecto de partículas de sangre. Pero si posteriormente tras haber llegado un nuevo comando de un peer y volver a simular dicho ataque resulta ser finalmente fallido habría que borrar ese efecto de partículas de sangre dando como resultado un *glitch* visual extraño. Por este motivo, en For Honor se hace diferencia entre dos tipos de eventos. El primer tipo son los eventos inmediatos, los cuales necesitan ser ejecutados inmediatamente después de que ocurra la acción. Como consecuencia, pueden verse cancelados si, tras volver a simular por la llegada tardía de un comando, esa acción no acaba ocurriendo, dando como resultado pequeños *glitches* visuales o sonoros. El otro tipo de evento son los eventos `Finalized` los cuales, a diferencia de los inmediatos, no son tan necesarios que se ejecuten justo después de haber ocurrido la acción por lo que el peer espera a que todos los comandos hayan llegado para ejecutar estos eventos y evitar así que ocurra ninguna resimulación que pudiese cancelarlos como en el caso anterior. Esto genera entre 1 y 2 segundos de retraso pero no produce ningún tipo de *glitch* visual o sonoro.

Ilustración 18 - Offset entre la posición de renderizado y la posición real de un personaje en For Honor

Por otro lado, Jennifer comenta la importancia de controlar las desincronizaciones entre *peers* ya que estas pueden acabar arruinando la partida si no se les da importancia. Es necesario tener una forma de detectar cuando estas suceden. Para ello, la arquitectura de For Honor implementa una técnica de monitorización la cual guarda en memoria todas las operaciones realizadas en todos los procesos de simulación desde que comienza la partida en una estructura llamada `Record Set`. Cada 100 milisegundos cada *peer* transmite su `Record Set` a un *peer* especial, previamente asignado, el cual se le conoce como "*Peer* de referencia". Este será el encargado de comparar todos los `Record Set` de todos los *peers*, incluido el del *Peer* de referencia, y validar que son idénticos. En caso de que esta validación fallase significaría que uno o varios *peers* estarían desincronizados. Ante esta situación, con el objetivo de recoger información para posteriormente analizarla y detectar la causa del problema, cada peer crea un fichero de lenguaje de marcado extensible (XML por sus siglas en inglés) donde guarda una traza con las simulaciones de los 512 últimos fotogramas. Este archivo puede llegar a pesar hasta 150 Megabytes. En este solo se incluyen los cambios ocurridos en el estado de juego en vez de la totalidad de este ya que en caso contrario el peso de este archivo sería mucho más grande. Por otra parte, en caso de desincronización, el *peer* de referencia le pide al *peer* que ha sufrido esta desincronización que le envíe todos los detalles de las operaciones de la simulación para poder compararlas y extraer las diferencias en otro archivo XML. Estas diferencias pueden ser desde una operación que no fue ejecutada en un *peer* o que un *peer* hizo una operación completamente distinta al resto, entre otras. Los archivos XML resultantes eran bastante difíciles de leer debido a su gran tamaño y al propio lenguaje XML, por ello, los desarrolladores crearon una herramienta capaz de visualizar las diferencias en las distintas operaciones de la simulación así como los detalles técnicos de cada *peer* (p.ej. el procesador, el mapa cargado…)

Este proceso descrito en el anterior párrafo era útil a la hora de detectar posibles errores de determinismo, aunque consumía muchos recursos puesto que crear trazas en archivos de tanto tamaño eran operaciones computacionalmente muy costosas e innecesarias fueras del

ámbito del desarrollo. Por otra parte, al exponer públicamente toda la información y detalles de la simulación da la posibilidad de que los jugadores puedan aprender a hacer trampas y hackear el juego. Debido a todos estos inconvenientes, se decide optar por una estrategia basada en *snapshots*. Como comenta Jennifer Henry en la ponencia [26] ahora, cada cierto tiempo, cada *peer* encapsula el estado de juego actual en una *snapshot* la cual se envía al *Peer* de referencia. Este es encargado de compararlas y buscar las diferencias para detectar posibles desincronizaciones. En caso de detectar alguna se realiza un procedimiento el cual consta de 3 pasos. En primer lugar, el *Peer* de referencia intenta que el *peer* o los *peers* desincronizados puedan reconciliarse con el resto recreando el estado de juego verídico a partir de la última *snapshot* válida. Si no lo consiguen, el *Peer* de referencia comprueba cuántos *peers* han sido afectados por la desincronización y en caso de ser una minoría se les expulsa de la partida. Si la desincronización afectó a más *peers* la partida tendría que ser cancelada. Con este nuevo método se resuelven los problemas del método previo descrito anteriormente, aunque como consecuencia, no se recopila ningún tipo de información para ser analizada y conocer las causas de la desincronización en la fase de *Debug*. Otra ventaja de este método basado en *snapshots* es que se les permite a otros jugadores unirse a mitad de partida. Según la implementación en For Honor se puede recrear el estado de juego actual de la partida únicamente a partir de la última *snapshot* y los comandos ocurridos desde dicha *snapshot* hasta la actualidad. Si por ejemplo un *peer* se une tras 10 minutos de partida solo tendría que realizar un promedio de 60 simulaciones para conseguir sincronizarse con el resto. Si esta funcionalidad se hubiese añadido con el anterior método, para un *peer* el cual se conecta tras 10 minutos de partida, este debería recibir todos los comandos desde el inicio de la partida para avanzar la simulación hasta el estado actual del juego. Esto hubiese tomado un total de 18000 simulaciones, un número mucho mayor que el obtenido en el método de *snapshots*, lo cual sería tremendamente costoso.

Por otra parte, el equipo de desarrollo se percató de la necesidad de optimizar la etapa de simulación por varias razones:

Por la latencia, cada *peer* recibía los paquetes con un retraso de 200 milisegundos aproximadamente, por lo que era necesario volver a simular aproximadamente un total de 8 estados de juego pasados en cada fotograma. La etapa de simulación está dividida en 3 fases como se muestra en la Ilustración 19. La primera de ellas es la fase de preparación donde se rebobina la simulación hasta el instante donde es necesario empezar a resimular. Luego está la fase de simulación la cual está dividida en bloques de 8 iteraciones, cada una de ellas formadas por un cálculo de físicas y un cálculo de *gameplay*, es decir, de las decisiones y acciones realizadas por el jugador. Por último, se encuentra la fase de finalización donde se liberan algunos recursos. Para que este proceso no afecte al rendimiento ni suponga caídas de FPS es recomendable que dure entre 30 y 40 milisegundos.

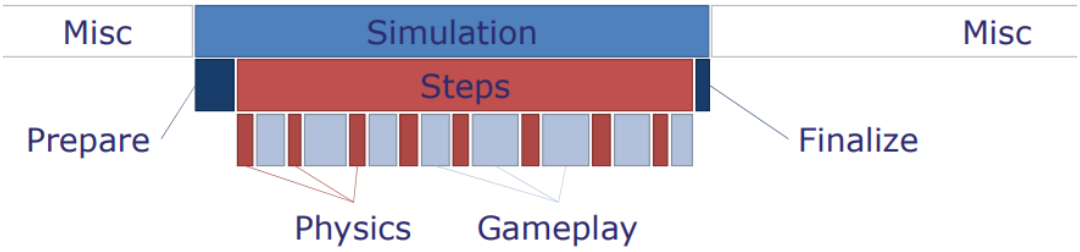


Ilustración 19 - Etapa de simulación en For Honor

Otra de las razones fue la naturaleza del juego. Al ser un videojuego multijugador debía de soportar las distintas actualizaciones de contenido las cuales añadían nuevos modos de juego, mapas… Si se quería que todo este nuevo contenido cupiese en el rango del tiempo de procesado de los 30 – 40 milisegundos que duraba cada etapa de simulación era necesario hacer algunas optimizaciones.

La primera optimización que se hizo fue trasladar el cómputo de la etapa de simulación a distintos hilos. Para ello se creó un doble *buffer* en donde cada entidad de la simulación poseía dos copias. Una la cual contenía el estado de dicha entidad justo antes de comenzar la simulación y otra la cual contenía el estado actualizado una vez acabase la simulación. La primera copia era *thread safe*, ya que se garantizaba su inmutabilidad y era de solo lectura. Esto permitía que los distintos hilos pudiesen realizar operaciones de lectura simultáneamente de forma segura. La segunda copia era la versión actualizada de la primera la cual se modificaba en el proceso de simulación. Pero en ella los distintos hilos no podían escribir de forma simultánea puesto que se podían generar condiciones de carrera. Para resolver este problema, se creó un sistema de mensajes con el objetivo de garantizar que los distintos hilos escribiesen en las copias actualizadas de las entidades de forma segura. Cada hilo creaba mensajes con la información que se quería modificar en la copia actualizada. Estos mensajes se almacenaban hasta que todos los hilos hubiesen acabado. Al final de la simulación un único hilo se encargaba de escribir uno por uno todos estos mensajes en las copias actualizadas.

Por otra parte, existía la posibilidad de que, debido a problemas de la red, los comandos tardasen algo más en llegar aumentando así el tiempo que sería necesario rebobinar y, como consecuencia, el número de iteraciones que habría que realizar en la resimulación. Como se ha comentado anteriormente este valor normalmente era de 8 iteraciones pero al no ser fijo este podría aumentar. Por ello, si se quería garantizar el procesado de esta simulación ante situaciones de picos de iteraciones sin que supusiese una caída de FPS, era necesario reducir el tiempo de procesado de otros elementos del juego como las animaciones o los efectos visuales dando como resultado una bajada de calidad en estos campos.

Una tercera optimización fue eliminar código o variables que no fuesen útiles ahorrando así capacidad de cómputo y memoria.

Estas optimizaciones fueron útiles, pero cuando se lanzó el nuevo modo de juego de For Honor llamado Breach, el cual triplicaba el número de entidades en el juego con respecto a

los modos anteriores, se realizaron algunas optimizaciones adicionales con el objetivo de evitar que, debido a este número creciente de entidades, el coste de procesamiento de todas ellas hiciese que bajasen los FPS.

En primer lugar se añade una especie de sistema de relevancia parecido al de otros títulos mencionados anteriormente como DOOM III o Quake III Arena. Este sistema ya no se basa en descartar todas aquellas entidades que estén fuera del campo de visión del jugador, sino en descartar aquellas que se encuentren en una fase del juego distinta a la del jugador. Esto es posible ya que el modo de Breach estaba basado en 3 fases diferenciadas. Cuando una fase nueva comienza, las entidades de la anterior fase dejan de ser relevantes y por lo tanto se eliminan temporalmente de la simulación. Por otra parte, las entidades de la nueva fase pasan a ser relevantes y a entrar a la simulación adoptando su estado inicial. De esta forma, se consigue no sobrepasar el número de entidades simultáneas que se procesan dentro de la simulación respecto a los modos de juego anteriores.

Como segunda optimización, la cual también está relacionada con el sistema de relevancia de las entidades, consistía en actualizar únicamente aquellas entidades que estuviesen cercanas a un jugador. Los desarrolladores se dieron cuenta que existen muchas entidades en el juego, como por ejemplo ballestas o puertas, las cuales la mayor parte del tiempo se encuentran en un estado `Idle` o por defecto esperando a que algún jugador interactúe con ellas. Como no es necesario actualizarlas mientras estén en dicho estado, se implementa un sistema basado en distancias donde si la entidad se encuentra lejos de los jugadores, esta pasa a un estado de `Sleep` y por lo tanto deja de actualizarse. Cuando un jugador se acerca, esta sale del estado de `Sleep` y vuelve a actualizarse. Esta optimización hizo que el coste en CPU se redujese hasta en un 90%

Otra optimización consistía en crear un tipo de *bot* llamado `Junior Bot` el cual se actualiza con una menor frecuencia que el resto. Hasta este momento, los *bots* tenían una tasa de actualización de 33 milisegundos, igual que la de los jugadores. Debido a que en el nuevo modo Breach de juego el número de *bots* crece, se ha tenido que rebajar esta tasa a 200 milisegundos, es decir, 6 veces menos que los *bots* normales resultando así en un menor coste a la hora de procesarlos.

Por último se aplicaron optimizaciones al código relacionado con las máquinas de estado finitas de los *bots* y las físicas.

3.3 Complementos teóricos

Desafíos y problemáticas de la red

A la hora de desarrollar un videojuego multijugador en línea, es necesario conocer los problemas que es necesario afrontar relacionados con la red.

Latencia:

El primero de ellos es la latencia, la cual se define como el tiempo que tarda la información en ser transmitida desde la capa de aplicación del cliente hasta la capa de aplicación del servidor [29]. Esta latencia es unidireccional ya que en el caso en el que se tuviera en cuenta tanto el

camino de ida como el de vuelta estaría haciendo referencia a otro término llamado RTT. Este valor también es conocido de forma informal como *ping*. Esta latencia va a depender de una serie de factores. En primer lugar va a estar afectada por el número de nodos intermedios de la red así como la velocidad de transmisión de cada uno de ellos. Cada vez que un paquete atraviese un nodo intermedio se verá afectado por un retardo nodal D_{nodal} el cual está formado por la suma de varios retardos [30, 31]:

1. Retardo de transmisión D_{trans}: Es el tiempo que tarda un nodo origen en enviar todos los bits de un paquete (constituidos por su cabecera H y su *payload* p) a través de la red. Este tiempo depende del tamaño del paquete $H + p$ y de la velocidad de transmisión b o más comúnmente conocida como ancho de banda, del nodo origen medida en *bits* por segundo (bps). $D_{trans} = {(H + p)}/{b}$
2. Retardo de propagación D_{prop}: Es el tiempo que tarda la información en viajar desde el nodo origen hasta el nodo destino. Este tiempo depende de la distancia d que tiene que recorrer la señal entre el origen y el destino y de la velocidad s de propagación del medio. $D_{prop} = {d}/{s}$
3. Retardo de procesamiento D_{proc}: Es el tiempo que tarda un nodo en procesar el paquete. Este tiempo depende de la velocidad de procesamiento del nodo.
4. Retardo de cola D_{queue}: Es el tiempo que un paquete se encuentra en un *buffer* a la espera de ser procesado por un nodo. Este tiempo depende de cómo de ocupado se encuentre dicho nodo en ese instante de tiempo.

Como resultado, el retardo nodal de un nodo intermedio puede calcularse con la fórmula $D_{nodal} = D_{trans} + D_{prop} + D_{proc} + D_{queue}$

Por otra parte, el tamaño de los paquetes también puede alterar la latencia. Esto es debido a que el tiempo de los retardos de transmisión de los nodos intermedios aumentará al depender del tamaño de los paquetes. Otro problema de los paquetes de gran tamaño es el de la fragmentación de paquetes. Cada nodo intermedio posee un tamaño máximo de paquete que podrá aceptar conocido como unidad máxima de transferencia (MTU por sus siglas en inglés). Cuando un nodo recibe un paquete de un tamaño mayor que su MTU tendrá dos opciones, (i) fragmentar el paquete o (ii) descartarlo [32]. Esto, en IPv4 va a depender del *bit* de la cabecera del paquete llamado `Don't Fragment` [33]. En caso de tener que fragmentarlo, se crearan múltiples fragmentos los cuales no se vuelven a reensamblar hasta que no llegue al host destino. Al fragmentarse un paquete, cada fragmento resultante contiene una copia de la cabecera original del datagrama IP mostrada en la Ilustración 20. Esta copia es parcial ya que cada fragmento modifica una serie de campos dentro de su copia como el `Fragment Offset`.

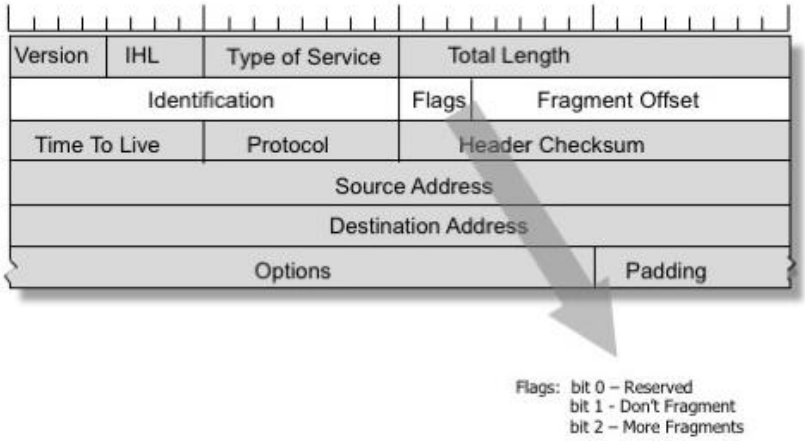


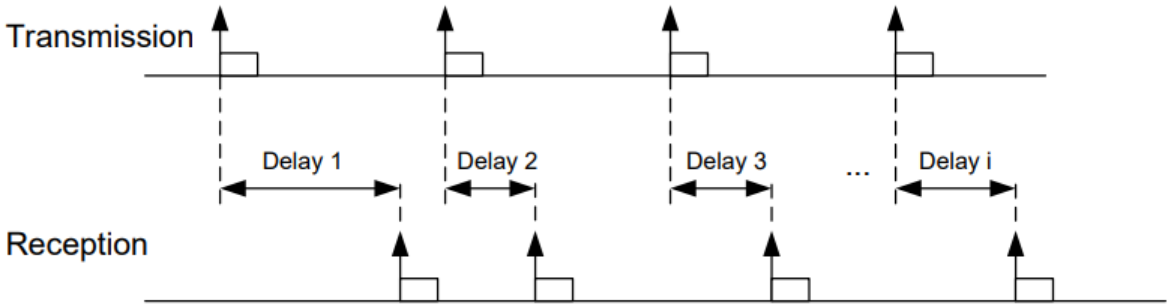
Ilustración 20 - Cabecera paquete IPv4

Este proceso de fragmentación trae consigo una serie de problemas. Por una parte, cuantos más fragmentos se generen, mayor es la posibilidad de que uno de ellos se acabe perdiendo. Si un fragmento se pierde y no llega al destino la totalidad del paquete se descarta. Por otra parte, este proceso de fragmentación conlleva una serie de retardos adicionales puesto que las operaciones de fragmentación y ensamblado del paquete ocupan un tiempo físico. Además, existe la posibilidad de que los fragmentos lleguen desordenados teniendo que ordenarlos en el host destino, y como consecuencia, aumentando el tiempo de ensamblado.

Este proceso de fragmentación únicamente ocurre en IPv4, puesto que en IPv6 el paquete directamente es descartado. En caso de IPv6 el host origen deberá de ser quien realice el proceso de fragmentación para evitar futuros problemas.

Jitter:

Otro problema de la red es aquel conocido como la fluctuación de la latencia (*jitter*) el cual se define como la variación de los retardos entre paquetes [29]. Esta puede ser causada por distintos retardos de cola como consecuencia de la congestión de la red. También puede ser causado por el fenómeno de *Bufferbloat* [34] en el que un nodo de la red posee un *buffer* excesivamente grande para almacenar los paquetes. Este fenómeno puede solucionarse usando políticas de manejo activo de cola (*Active Queue Management* en inglés, AQM por sus siglas en inglés) [35]. En la Ilustración 21 se muestra una representación gráfica del *jitter* así como la fórmula para calcularlo [29].



$$IPDV = \frac{\sum \left| delay_{i} - delay_{i-1} \right|}{n - 1}$$

Ilustración 21 - Representación gráfica del jitter

El *jitter* se traduce en una mayor latencia y en ocasiones en pérdida de paquetes. Esto es debido a que en muchas ocasiones esta variación ha sido tan grande que el paquete ha llegado demasiado tarde quedando así obsoleto (p.ej. porque un paquete más nuevo ha llegado antes). Esto puede solucionarse usando un Buffer de jitter.

Pérdida de paquetes:

Otro problema es la pérdida de paquetes el cual puede ser causado por diferentes factores. Anteriormente se ha comentado que debido al gran tamaño de paquetes, la técnica de fragmentación de paquetes aumenta la posibilidad de perder dicho paquete fragmentado. Por otra parte, también se ha comentado que debido al *jitter*, los paquetes obsoletos pueden ser automáticamente descartados por el host destino. Otra posible causa puede ser debido a la congestión. Esta hará que los *buffers* de los nodos intermedios se saturen haciendo que los paquetes entrantes sean automáticamente descartados por falta de capacidad. Esto no solo puede ocurrir cuando el *buffer* se satura, sino que debido a las AQMs de los nodos, algunos paquetes pueden descartarse antes de que el *buffer* se llene.

Debido a que la pérdida de paquetes es un problema inevitable, muchos juegos multijugador implementan técnicas como Extrapolación de entidades en el cliente con el objetivo de ocultar estas pérdidas.

Arquitecturas cliente servidor

A la hora de desarrollar un videojuego multijugador, una de las primeras decisiones que han de tomarse es el tipo de arquitectura en red que se va a usar. A pesar de que existen varias opciones cada una de ella presenta ventajas e inconvenientes respecto a temas como la escalabilidad, la latencia, la seguridad o la dificultad del desarrollo del proyecto. Esta elección va a depender de varios factores entre los cuales se encuentran el tipo de juego que se vaya a desarrollar o el presupuesto disponible [36].

Una de las arquitecturas más usadas es el modelo de cliente servidor. En este, existe un único servidor encargado de ejecutar toda la lógica principal del juego [22]. En esta arquitectura es muy común que el servidor sea autoritativo, lo que significa que su estado de juego es considerado correcto [15]. Por otra parte, debido a esta autoridad, el servidor es el que toma todas las decisiones en el juego. Por ejemplo si un jugador dispara una bala, será el servidor quien determine si esta ha impactado con algo o no. Al servidor se conectan diferentes clientes los cuales actúan como terminales "tontas" [22] debido a que en principio, las únicas tareas que realizan es recoger los comandos por parte del jugador para enviarlos al servidor y cuando lleguen las actualizaciones del estado de juego, renderizarlo por pantalla. Por otra parte, el servidor se encarga de recibir estos comandos del resto de clientes, procesarlos avanzando el estado del juego y enviando este de vuelta a los clientes.

Dentro de esta topología cliente servidor, existe una variante llamada Host o Listen Server. Esta variante hace que uno de los clientes actúe a su vez también como servidor [37]. Como consecuencia se consigue reducir los costes de alquiler o mantenimiento de infraestructuras, lo cual es una ventaja, puesto que se usa el *hardware* de uno de los equipos de los clientes. Aunque la máquina elegida para hacer de *host*, es decir actuar tanto como de cliente como de servidor de forma simultánea, deberá de ser lo suficientemente potente para poder realizar las tareas tanto de servidor como cliente sin que su rendimiento se vea gravemente afectado. Además esta deberá de ser capaz de soportar todo el tráfico de red. En especial, deberá de tener una buena velocidad de subida de paquetes para poder enviar *snapshots* al resto de clientes. Como punto negativo, al usar esta variante de *host* el juego queda expuesto a posibles trampas ya que el servidor, al convertirse en *host*, deja de ser fiable.

Por otra parte, los estados de juego locales de cada uno de los clientes deben mantenerse actualizados con respecto al estado de juego en el servidor para evitar que estos acaben desincronizándose [38]. A este proceso se le conoce como replicación. Existen varios tipos, aunque por la naturaleza de la arquitectura cliente servidor la más común es la replicación pasiva. En la replicación pasiva solo una máquina, el servidor, es responsable de todo el estado de juego. Los clientes envían sus comandos únicamente al servidor, este los procesa y envía las actualizaciones del estado de juego de vuelta a todos los clientes (En forma de *broadcast*) En la replicación pasiva, cuando se crea un objeto dentro de la simulación, se crea una réplica maestra [39] la cual la posee aquella máquina con autoridad sobre dicho objeto. En la arquitectura cliente servidor con este último siendo autoritativo las réplicas maestras se encuentran en el servidor. Por cada objeto solo puede existir una única réplica maestra. Tras ello, el resto de clientes crean una réplica esclava, la cual sería una representación local de la réplica maestra. Este sistema de maestro-esclavo permite ejecutar en una entidad distintos comportamientos dependiendo de si es una réplica maestra o una réplica esclava como por ejemplo técnicas específicas para ocultar la latencia.

Este tipo de replicación tiene una serie de ventajas [38] ya que es especialmente robusta ante desincronizaciones y permite implementar medidas para evitar *hackers* y tramposos. Aunque como desventajas el servidor debe de ser capaz de soportar toda la carga de red y de procesamiento del estado de juego. Además, en caso de haber únicamente un servidor, si este falla todo el estado de juego se vería afectado. Otro punto negativo es el de la escalabilidad, puesto que cuantos más jugadores haya conectados más potente deberá de ser el servidor.

Funkhouser propone un sistema llamado RING [40] el cual usa un *cluster* (conjunto de servidores) los cuales serán encargados de procesar el estado de juego y enviar actualizaciones a los clientes. Además, este sistema implementa algoritmos de gestión de la relevancia con el objetivo de reducir el número de mensajes enviando solo aquellos que sean relevantes para cada cliente.

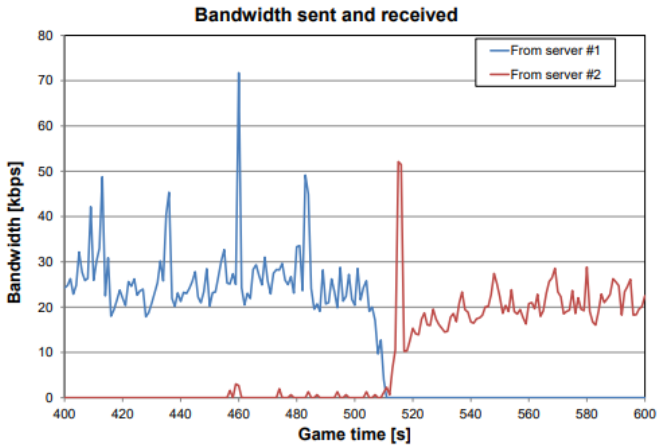
Una forma de realizar la comunicación entre cliente y servidor es a través de las RPCs [41] las cuales permiten ejecutar código en otra máquina remota pudiendo definir unos parámetros de entrada opcionales. Estas suponen un proceso de comunicación bidireccional basado en mensajes. Además, las RPC intentan seguir el principio de transparencia haciendo que se parezcan lo máximo posible a las llamadas locales, aunque existen una serie de diferencias como por ejemplo la mayor latencia de las RPC o su mayor dificultad a la hora de encontrar errores. Actualmente, muchas librerías encargadas de dar soporte multijugador como **Netcode For GameObjects** [42] o **Photon** [43] utilizan llamadas RPCs para realizar la comunicación entre máquinas.

Debido a que estas llamadas hacen uso de una conexión en red y de una máquina remota que actúe como destino es difícil conocer el resultado de estas operaciones con certeza. Como ya se comentó en el apartado anterior, los paquetes que viajan a través de la red son susceptibles a sufrir ciertos problemas como retardos o pérdidas. Como consecuencia es importante tener en cuenta que estas llamadas puedan ejecutarse una vez, varias veces o ninguna vez en la máquina destino. Debido a esto es necesario que el código sepa manejar estas anomalías y sea tolerante a los problemas de la red.

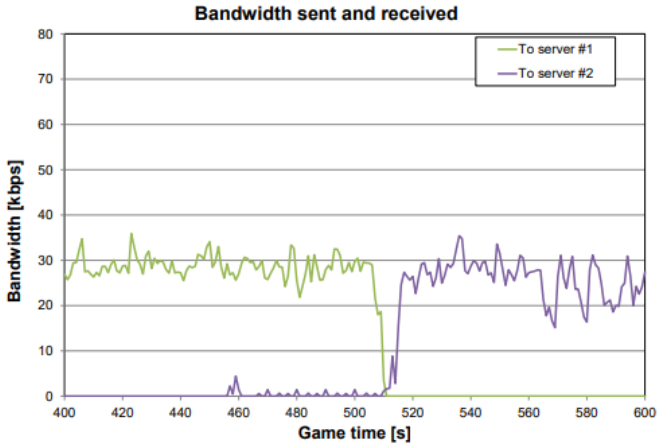
Por otra parte, los servidores de juego pueden ser controlados de varias formas según las políticas de las compañías [29]:

1. En primer lugar, el programa encargado de levantar el servidor puede darse junto con el juego, el cual puede ser instalado por el jugador. En estos casos el jugador tendría el control del servidor y podría ser capaz de levantar un servidor dedicado o de usarlo como host. Esto permitiría crear partidas mediante LAN con unas latencias mínimas. Esto haría que los desarrolladores no tuviesen control sobre estos servidores pero tampoco tuviesen que pagar por su mantenimiento. Esto está permitido en juegos como **Warcraft 3** o **Counter Strike 1**.
2. Otra forma sería incluir, al igual que en el primer caso, el servidor dentro del propio juego pero sin que el jugador tuviese control sobre este. Esto haría que uno de los clientes actuase tanto como cliente como servidor pero este sería elegido por los desarrolladores mediante algún tipo de selección (p.ej. aquel cliente con mejores condiciones de red). Este proceso es invisible para los jugadores, ya que estos no saben quien está actuando como servidor. Al igual que en el anterior caso, los desarrolladores no pagan por mantener estos servidores puesto que se usa el *hardware* de alguno de los clientes. Pero este modelo trae una serie de desventajas. Por una parte, el cliente que a su vez actúa como servidor tendría una clara ventaja puesto que su latencia

respecto al servidor, es decir respecto a él mismo, sería nula. Por otra parte si el cliente que actúa como servidor abandona la partida se ejecutaría un nuevo proceso de selección para elegir otro cliente. Esto ocasionaría un parón en la partida apreciable por todos los jugadores mientras que se realiza el proceso de mitigación de servidor (*server mitigation*). En la Ilustración 22 se puede apreciar este parón durante el proceso de mitigación de servidor del Call Of Duty de PlayStation 2 entre el Segundo 500 y el 520.



(a)



(b)

Ilustración 22 - Tráfico de red durante el proceso de mitigación de servidor en Call Of Duty

3. Por último algunos videojuegos como el World Of Warcraft poseen servidores totalmente controlados por los desarrolladores. Esta es la tendencia dominante en los juegos de hoy en día. Esto trae como desventaja el coste extra de alquilar y mantener dichos servidores pero a cambio pueden controlar y prevenir en cierta medida a los *hackers* o implementar modelos de negocio vendiendo objetos del juego a cambio de dinero real. Por otra parte, mediante el uso de ingeniería inversa, se pueden crear, mediante técnicas de emulación, versiones privadas de los servidores de juego.

Aunque debido a las constantes actualizaciones por parte de los desarrolladores estas emulaciones tienden a dejar de funcionar y quedar obsoletes en poco tiempo.

Técnicas de compensación de latencia

Las técnicas de compensación de latencia son algoritmos de *software* ejecutados en el cliente, en el servidor o en ambos los cuales tienen como objetivo minimizar los efectos de la latencia en los jugadores [44]. Cada una de las técnicas puede tener un impacto sobre la consistencia o la responsividad del juego. La consistencia hace referencia a las diferencias entre el estado de juego del cliente y el del servidor mientras que la responsividad hacer referencia al retraso que existe desde que el jugador introduce un comando hasta que este se ve reflejado en el estado de juego [44]. Dependiendo de la técnica este impacto puede ser positivo o negativo y normalmente suelen mejorar un aspecto a cambio de empeorar el otro. En la Ilustración 23 se puede ver una representación gráfica de la relación entre una técnica de compensación de latencia con el impacto en la responsividad y la consistencia. Aquellas técnicas que consigan aproximarse más al punto en el que nacen ambos ejes serán las que mayor impacto positivo tendrán sobre estos dos aspectos.

Ilustración 23 - Relación entre la responsividad y la consistencia de las técnicas de compensación de latencia

Por otra parte, la implementación de técnicas capaces de compensar la latencia mejora tanto el rendimiento como la calidad de la experiencia de los jugadores en el juego [45]. En las siguientes dos ilustraciones, sacadas del informe [45], se aprecian estas consecuencias. En la Ilustración 24 se puede ver cómo tanto jugadores experimentados, representados por el color azul, como novatos, representados por el color rojo, empeoran sus puntuaciones en el CS:GO debido a la latencia. Este impacto se nota más en el grupo de jugadores experimentados. Además, en la Ilustración 25 se ve como la calidad de su experiencia en el juego se ve mermada por culpa de la latencia, una vez más teniendo mayor impacto en aquellos jugadores experimentados.

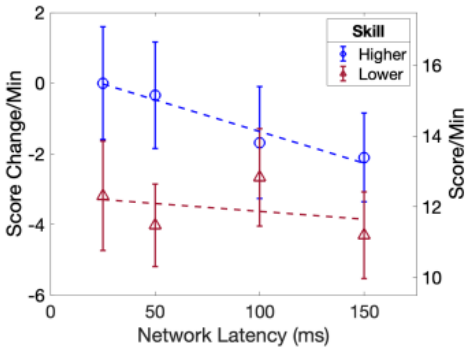


Ilustración 24 - Relación entre la puntuación y la latencia en CS:GO

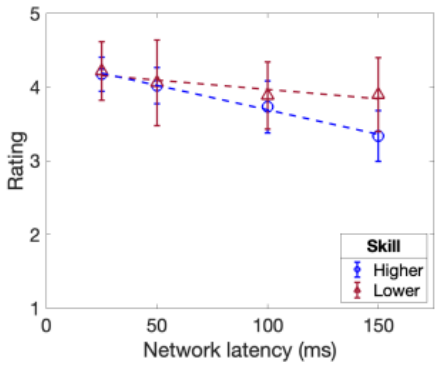


Ilustración 25 - Relación entre la calidad de la experiencia y la latencia en CS:GO

Otro ejemplo de esta evidencia fue el de Quax et al. [46] quienes realizaron un estudio sobre cómo distintos valores de latencia y *jitter* afectaban a los jugadores de Unreal Tournament. Observaron que cuanto mayor era la latencia, peor era la calidad de la experiencia de juego. En la Ilustración 26 se puede ver cómo los mismos jugadores obtuvieron puntuaciones más bajas en escenarios con peores condiciones de red (representados por diamantes unidos por líneas continuas) en comparación con escenarios con mejores condiciones de red (representados por triángulos unidos por líneas continuas).

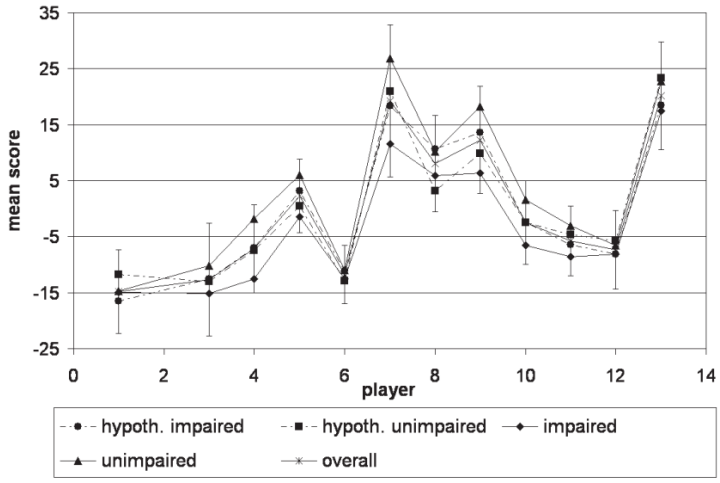


Ilustración 26 - puntuación jugadores afectados y no afectados por la latencia en Unreal Tourtnament

Mark et al. [44] proponen una jerarquía de técnicas de compensación de latencia representada en la Ilustración 29. En los próximos apartados, se explicarán algunas de ellas.

En primer lugar, existe el grupo de técnicas de retroalimentación (Feedback), cuyo objetivo es proporcionar una retroalimentación tanto auditiva como visual al jugador basada en la latencia. Por este motivo, estas técnicas son exclusivas del cliente. Un ejemplo sería la técnica de exposición de la latencia en la que se proporciona un indicador visual al jugador sobre la cantidad de latencia desde el cliente al servidor. Esta puede exponerse de forma numérica como se muestra en la Ilustración 27 o mediante el uso de iconos como en la Ilustración 28. Este grupo de técnicas mejoran la responsividad sin afectar negativamente a la consistencia.

Ilustración 27 - Exposición de la latencia en Call Of Duty: Modern Warfare II

Ilustración 28 - Iconos para exponer información sobre la calidad de la conexión en Rocket League

El segundo grupo es el de técnicas de predicción, cuyo objetivo es el de predecir nueva información en base a la que ya se dispone. Una técnica de este grupo es la Extrapolación de entidades en el cliente. Este grupo de técnicas ofrecen una mejora en la responsividad a cambio de empeorar la consistencia.

Por otra parte, el tercer grupo de técnicas se encarga de manipular el tiempo virtual alterando el estado del juego con el objetivo de procesar o resolver distintas acciones del jugador. Un ejemplo de este grupo es la técnica de *Time Warp*, usada para retroceder el estado de juego a un instante en el pasado en el que se produjo una acción del jugador con el objetivo de aplicar dicha acción, compensando así la latencia, y volviendo a avanzar el estado de juego hasta la actualidad.

El último grupo principal que se propone es el del ajuste del mundo el cual tiene el objetivo de modificar el estado de juego para reducir la dificultad con configuraciones con menos latencia. Un ejemplo de este grupo son las técnicas de control de asistencia, como la técnica de *Sticky Targets* (Objetivos pegajosos) la cual reduce la cantidad de movimiento del cursor cuando este se encuentra cerca de un objetivo. Bateman, Gutwin et al. [47] ponen a prueba la eficacia de distintas técnicas de control de asistencia (entre ellas la de *Sticky Targets*) en el género de disparos en primera persona.

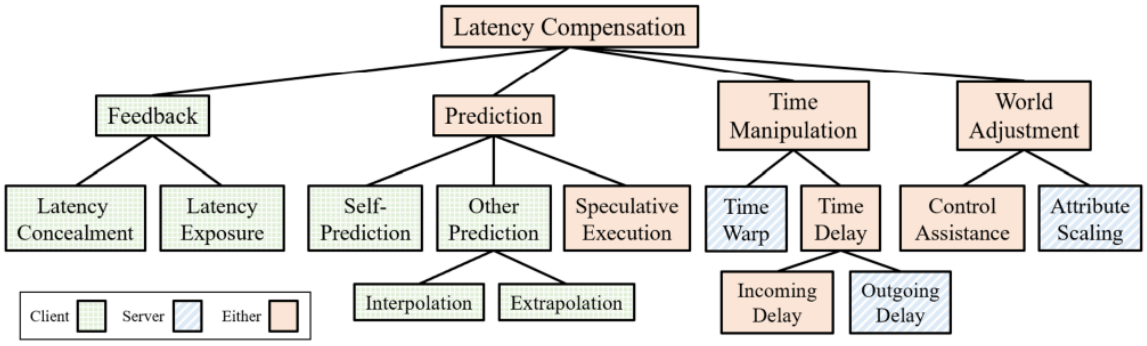


Ilustración 29 - Jerarquía de técnicas de compensación de latencia

Predicción de comandos en el lado del cliente

La predicción de comandos en el lado del cliente es una de las técnicas para reducir el problema de la latencia, la cual consiste en llevar a cabo el movimiento del cliente de forma local asumiendo que el servidor aceptará los comandos generados por dicho cliente [22]. Sin este método, el siguiente proceso se llevaría a cabo: (i) el cliente generaría comandos (ii) los cuales serían transmitidos al servidor. (iii) Este calcularía el siguiente estado de juego y (iv) lo enviaría a todos los clientes para que estos pudiesen (v) renderizar la escena con el nuevo estado de juego [29]. Pero esto generaría un intervalo de tiempo en el que el cliente estaría esperando a que llegase el mensaje por parte del servidor para poder renderizar el siguiente estado. Por ello, como la mayoría de los clientes poseen un motor de videojuegos integrado (con físicas, colisiones…), al implementar la predicción el cliente podrá calcular el movimiento de forma local proporcionando una retroalimentación instantánea para el jugador [44], y como consecuencia, aumentando la responsividad del juego.

Este método posee un inconveniente ya que estas operaciones de predicción pueden generar resultados distintos respecto a los del servidor generando inconsistencias en el estado de juego. Cuando los clientes implementan la predicción, estos dejan de ser terminales tontas (*Dumb Clients*) [22]. El cliente toma un control total de su simulación, al igual que ocurre en las arquitecturas P2P sin un servidor central. Pero esto únicamente ocurre de forma temporal puesto que el servidor sigue teniendo la autoridad. Si los resultados de la simulación enviados desde el servidor no coinciden con los obtenidos por el cliente este deberá de realizar una corrección para seguir sincronizado con el servidor mediante una técnica conocida como reconciliación con el servidor. Incluso si el cliente quisiera hacer trampas y modificar su estado de juego solo él se vería afectado por estas acciones, pero ni el servidor ni el resto de clientes serían perjudicados. Debido a la latencia, el cliente deberá de esperar mínimo un tiempo igual al RTT de la conexión con el servidor hasta recibir los resultados verídicos. Esta corrección puede ocasionar cambios perceptibles en la posición del jugador.

Para llevar a cabo la reconciliación con el servidor, el cliente antes de enviar los comandos al servidor para que sean procesados este los guarda en un *buffer* local. Cada comando posee el instante de tiempo en el que fue generado así como un identificador y la información de *inputs*. Cuando el servidor envía *snapshots* al cliente cada una de estas *snapshots* no solo contiene la información acerca del estado del jugador en el instante de tiempo en el que se generó dicha *snapshot*, sino también el identificador del último comando que fue procesado [37]. Tras recibir esta información, el cliente verifica si la predicción local que hizo en el pasado dio los mismos resultados que la contenida en la *snapshot* del servidor. En caso de diferir sería necesario realizar una corrección. Para que esta no quede muy brusca, se podrían usar técnicas de suavizado exponencial.

Por otra parte, el proceso de predicción constará de varios pasos. En primer lugar, es necesario determinar cual fue el último comando simulado y enviado por parte del servidor. Una vez encontrado, se volverá a simular el estado del jugador partiendo de dicho comando hasta el actual. Este método puede generar problemas ya que si ejecutamos los mismos comandos una y otra vez, es posible que se invoquen eventos como por ejemplo sonidos, animaciones entre otros de forma repetida. Para solucionar esto, cada comando puede tener una *flag* que indique si es la primera vez que se ejecuta o no. De esta forma, la simulación se puede

configurar para que únicamente se invoquen este tipo de eventos la primera vez que se ejecuta cada comando.

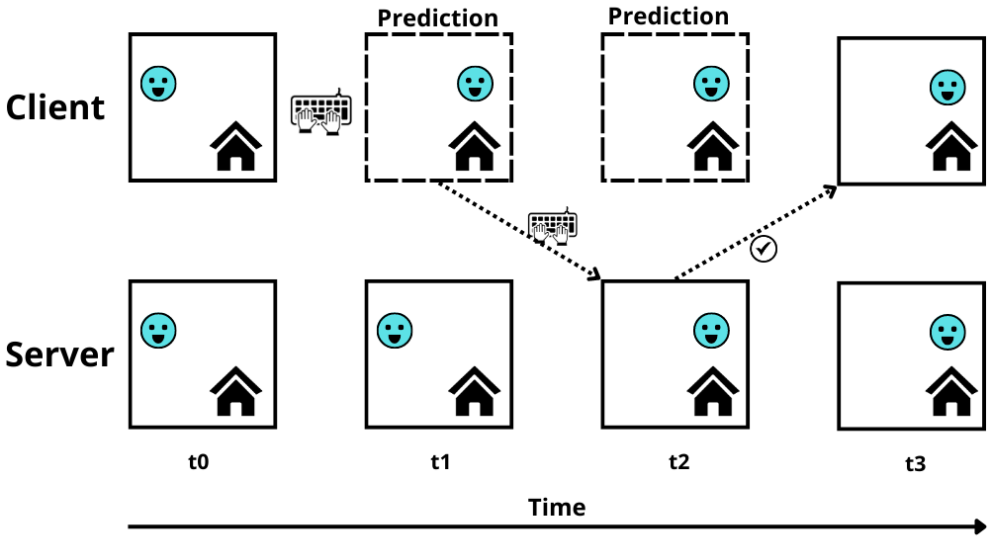


Ilustración 30 - Ejemplo de predicción en el lado del cliente

En la Ilustración 30 se muestra un ejemplo de predicción de comandos en el lado del cliente. En él, se representa el estado del juego de un cliente y un servidor en cuatro momentos diferentes de la misma partida. En el instante t0 el personaje controlado por el cliente se encuentra en la parte superior izquierda del mapa. En el instante t1 el cliente genera un *input* (p.ej. ha presionado la tecla D) con la intención de mover al personaje hacia la derecha. Este *input* se envía al servidor para que realice la simulación y envíe los resultados de vuelta al cliente. Aquí es donde ocurre la predicción ya que, en vez de esperar a que lleguen los resultados del servidor, el cliente lleva a cabo el mismo cálculo para predecir y aplicar el siguiente estado del personaje dando así la sensación de instantaneidad. En el instante t2 el servidor recibe los *inputs* enviados por el cliente y simula el estado del juego. Una vez acabada la simulación envía de vuelta los resultados obtenidos al cliente. Por último, en el instante t3 el cliente recibe los resultados oficiales de la simulación realizada en t1. En este punto, pueden darse dos escenarios: Que los resultados de la simulación en el cliente y en el servidor hayan sido iguales, o que estos hayan diferido. En caso de darse la segunda opción se estaría creando un error de consistencia entre el cliente y el servidor por lo que, como el servidor siempre es el que posee la autoridad y la veracidad de los datos, habría que forzar al cliente a hacer coincidir su estado del personaje con el estado recibido del servidor mediante la técnica de reconciliación.

Para poder obtener mejores resultados en esta técnica y mejorar la consistencia entre los distintos nodos de la red, es deseable que el cálculo de la simulación sea determinista. Esto quiere decir que dada la misma entrada, en este caso formada por el estado actual y el *input* del cliente, siempre se producirá el mismo estado de salida. Como contrapartida, esta consideración puede aumentar la complejidad del desarrollo ya que habría que tratar temas

como la aleatoriedad o los errores de precisión producidos por las operaciones de coma flotante.

Interpolación de entidades en el cliente

La interpolación de entidades es otra técnica de compensación de latencia encargada de determinar el estado de objetos controlados por otros jugadores o inteligencias artificiales a partir de un estado del juego, ligeramente en el pasado, del que se tiene más información [44]. Esta técnica es normalmente usada con el objetivo de proporcionar una mayor fluidez visual en el cliente en aquellos casos en los que este tenga una tasa de actualización mayor que la tasa de envío de paquetes del servidor. En caso de no aplicarse el cliente vería como los distintos objetos se teletransportan distancias pequeñas cada vez que llega una actualización por parte del servidor, lo cual puede acabar afectando a la calidad de la experiencia del usuario [48].

En la Ilustración 31 se puede apreciar un ejemplo sacado del Source Engine de esta técnica [49]. En ella el cliente recibe 20 *snapshots* por segundo por lo que la diferencia de tiempo entre dos *snapshots* consecutivas es de 50 milisegundos. Por otra parte el cliente almacena dichas *snapshots* en el *buffer* durante un periodo de 100 milisegundos, lo cual equivale a dos *snapshots*, tras los cuales comenzará la interpolación. Esto permite que en caso de que una *snapshot* llegue a perderse, el cliente siga teniendo una *snapshot* adicional para seguir interpolando. En el ejemplo, la última *snapshot* fue recibida por el cliente en el tiempo 10.30. Actualmente este cliente se encuentra en el tiempo 10.32 pero, debido a que va con 100 milisegundos de retraso, el tiempo de interpolación será 10.22. Esto hará que el cliente interpole entre las *snapshots* 340 y 342. Si, debido a la pérdida de paquetes, la *snapshot* 342 no hubiese llegado el cliente hubiese podido seguir interpolando entre las *snapshots* 340 y 344.

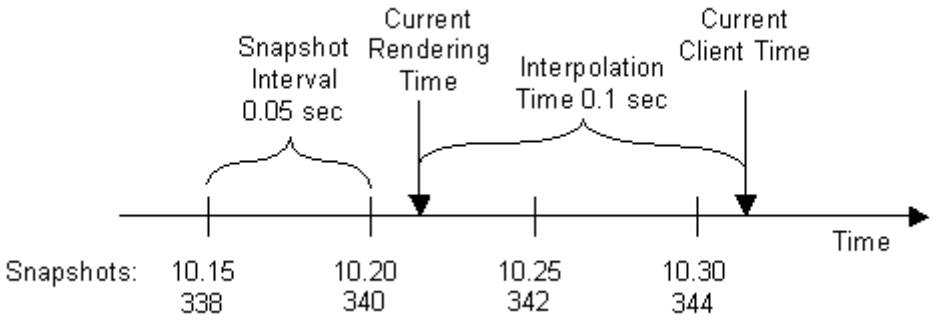


Ilustración 31 - Proceso de interpolación en el lado del cliente

Extrapolación de entidades en el cliente

La extrapolación es una técnica de compensación de latencia la cual tiene como objetivo predecir estados futuros de objetos controlados por otros jugadores asumiendo que sus comportamientos actuales seguirán invariables [44]. A este método también se le conoce como *Dead Reckoning*. Esta técnica es usada cuando, debido a las problemáticas de la red, el

Buffer de jitter usado en las técnicas de interpolación se queda vacío sin más estados entre los que interpolar [22]. En ese caso se generará una incertidumbre en el estado de la entidad la cual deberá de ser manejada. Para ello, se asume que la entidad continúa moviéndose en la misma dirección que tenía en el último estado recibido por parte del servidor. En base a esta asunción se predice el siguiente estado.

El éxito de los resultados de las técnicas de extrapolación está estrechamente relacionado con el tipo de videojuego en el que se aplican. En los videojuegos de carreras de coches, la maniobrabilidad que posee el jugador respecto al movimiento del coche es relativamente pequeña [48]. La posición del vehículo a altas velocidades depende en gran parte de la velocidad, dirección y posición previa. Si por ejemplo un vehículo se mueve a 100 metros por segundo, es casi seguro que tras 1 segundo este habrá avanzado 100 metros. El jugador puede acelerar, decelerar o girar pero la diferencia de la nueva posición respecto a la predicha sería pequeña. A no ser que el vehículo colisione contra algún objeto su velocidad no se va a ver bruscamente afectada. Por otra parte esto también puede aplicarse a videojuegos con movimientos a bajas velocidades como un videojuego de guerra de barcos. En contraposición, otros videojuegos como los de disparos en primera persona poseen un movimiento mucho más cambiante donde los jugadores pueden girar de forma prácticamente inmediata haciendo que el movimiento sea mucho menos predecible que en el anterior caso [22]. Esto hará que la diferencia entre la estimación de la incertidumbre y la real del servidor en el proceso de extrapolación sea cada vez más grande a medida que aumenta el tiempo de uso de la extrapolación. Una forma de reducir esta diferencia es estableciendo un tiempo máximo de extrapolación. Al hacer esto, si el tiempo que se llevan usando estas técnicas para predecir futuras posiciones supera este umbral, se deja de predecir. Con esto se consigue disminuir la magnitud de los saltos generados por las correcciones cuando se recibe la verdadera posición por parte del servidor.

En caso de no recibir una *snapshot* por parte del servidor en un tiempo razonable y de haber alcanzado el umbral del tiempo máximo de extrapolación será necesario realizar alguna de las siguientes acciones [37]:

1. Forzar la desconexión del cliente afectado.
2. Congelar el juego del cliente afectado y esperar a que este vuelva a recibir mensajes del servidor para reanudar el juego. En caso de no tener éxito se le desconectaría de forma forzosa
3. Desconectar al cliente afectado e inmediatamente después intentar volverlo a conectar.

En caso de volver a recibir *snapshots* por parte del servidor tras estar aplicando la extrapolación se dejaría de extrapolar para comenzar a interpolar con la nueva información recibida. Esta transición de nuevo a la interpolación debe de ser lo más fluida posible para evitar que afecte a la calidad de la experiencia del jugador. En muchos casos esta extrapolación habrá generado errores entre la posición actual del objeto y la posición en la que debería de estar, los cuales deberán de ser corregidos. Si este error es muy grande la mejor forma es realizar un desplazamiento brusco e instantáneo a la posición correcta recibida en la *snapshot*.

Por el contrario, si la posición solo difiere un poco visualmente quedará mejor si se va corrigiendo suavemente a lo largo del tiempo aplicando un porcentaje de la corrección en cada fotograma. A este método se le conoce como suavizado exponencial (*exponential smoothing*). Staken y AlRegib realizaron un estudio donde se describían y evaluaban varios algoritmos de suavizado exponencial en entornos virtuales colaborativos para estudiar su eficacia [50]. Por último, si el error es mínimo, la corrección puede obviarse.

Existen varios algoritmos capaces de predecir futuros estados a partir de los actuales [51]. La forma más simple de aplicar el *Dead Reckoning* es mediante la ecuación lineal del movimiento rectilíneo uniforme. Esta permite calcular una posición en el futuro en base a la posición y velocidad actuales y un incremento en el tiempo. De esta forma si se tiene los siguientes *inputs*:

$r[k]: Posición\ en\ el\ instante\ k$
$v[k]: Velocidad\ en\ el\ instante\ k$
$i: Incremento\ de\ tiempo$
$s: tiempo\ entre\ dos\ actualizaciones\ consecutivas$

Se podrá obtener la posición en el instante $k + i$ de la siguiente forma:

$$r[k+i] = r[k] + tv[k], t = is$$

Esta fórmula se podría ir mejorando con información adicional como por ejemplo si se dispone de la aceleración lineal o la velocidad angular entre otros. En caso de que $i = s$ la posición obtenida en esta fórmula sería la predicha en el mismo instante de tiempo de la siguiente actualización por parte del servidor. Por lo que tras haber obtenido esta nueva posición, se podría aplicar la técnica de interpolación vista en el apartado anterior para suavizar la transición entre ambos estados a cambio de introducir algo de error debido al retraso.

Dentro de estas técnicas de interpolación existen varias técnicas. Por una parte, la forma más simple de interpolación es usando un algoritmo de interpolación lineal [52]. Esta crea recorridos fluidos entre dos puntos, pero el recorrido final posee discontinuidades mostradas en la Ilustración 32 en forma de picos.

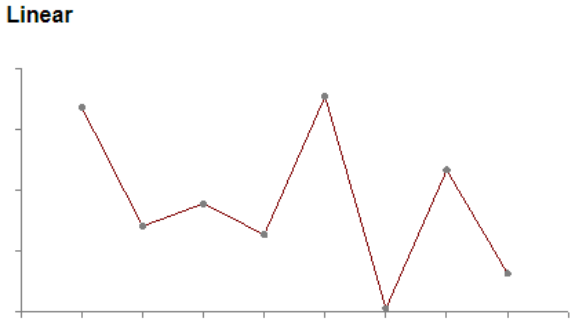


Ilustración 32 - Camino generado con el algoritmo de interpolación lineal

Otras soluciones que resuelven el problema de la interpolación lineal son algoritmos capaces de definir curvas [53], como por ejemplo las curvas de Bezier, Hermite… Estas son capaces de crear movimientos fluidos y caminos sin discontinuidades. Aunque si el objeto realiza una gran cantidad de cambios de dirección, estas tienden a crear pequeñas oscilaciones. Por otra parte, si la tasa de actualización no es a intervalos constantes estas oscilaciones se agravarán aún más.

Un algoritmo capaz de minimizar estas oscilaciones es el de *Projective Velocity Blending* (Mezcla de velocidades proyectadas). Este algoritmo es capaz tanto de interpolar, como de realizar la predicción al mismo tiempo. A diferencia de los enfoques anteriores, en este algoritmo se calculan dos proyecciones, una a partir del último estado conocido y otra a partir del estado actual. Una vez calculadas se realiza una interpolación lineal entre ambas. Además, se realiza otra interpolación lineal de las velocidades de ambas proyecciones. A continuación, se muestran las fórmulas matemáticas de este algoritmo.

En primer lugar se calcula la velocidad interpolada V_b resultante de aplicar la fórmula de la interpolación lineal entre la velocidad actual V_0 y la última velocidad conocida V'_0 en el intervalo de tiempo normalizado $\hat{T}$.

$$V_b = V_0 + (V'_0 - V_0)\hat{T}, Lerp(V_0, V'_0, \hat{T})$$

El intervalo de tiempo $\hat{T}$ es un intervalo normalizado el cual tiene valor 0 en el instante de tiempo de la última actualización del servidor y 1 en el instante de tiempo en el que la siguiente actualización del servidor debería llegar.

Una vez calculada la velocidad interpolada esta se usará para calcular las posiciones de ambas proyecciones donde T_t es el tiempo transcurrido desde la última actualización por parte del servidor. En este ejemplo, se está usando la fórmula del movimiento rectilíneo uniformemente acelerado $P = P_0 + V_0 t + \frac{1}{2}at^2$

$$P_t = P_0 + V_b T_t + \frac{1}{2}A'_0 T_t^2$$

$$P'_t = P'_0 + V'_0 T_t + \frac{1}{2}A'_0 T_t^2$$

Por último, una vez calculadas ambas proyecciones, se volverá a aplicar la fórmula de la interpolación lineal para obtener la posición final interpolada Q_t.

$$Q_t = P_t + (P'_t - P_t)\hat{T}, Lerp(P_t, P'_t, \hat{T})$$

En la Ilustración 33 se muestra un ejemplo de los resultados obtenidos con el algoritmo de *Projective Velocity Blending* sacado de la ponencia de Delbosc [54].

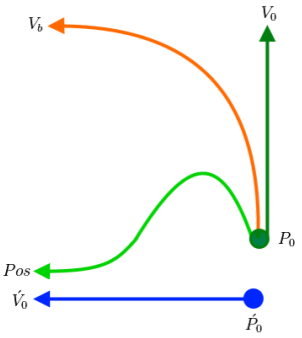


Ilustración 33 - Representación gráfica del algoritmo de Projective Velocity Blending

Por otra parte, en la Ilustración 34 se muestra cómo el algoritmo de *Projective Velocity Blending* minimiza las oscilaciones creadas con los algoritmos de creación de curvas (En este caso se compara con el algoritmo de *splines* cúbicos de Bezier) [53].

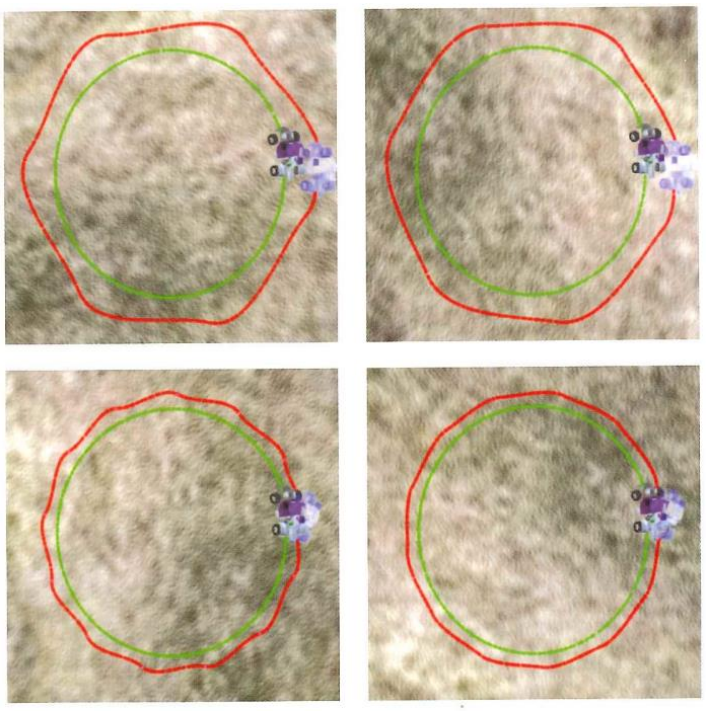


Ilustración 34 - Splines cúbicas de Bezier (izquierda) comparado con Projective Velocity Blending (derecha), usando aceleración (arriba) y sin usarla (abajo)

Un problema que presentan todos los algoritmos anteriores es la detección de colisiones. Puede darse el caso en el que un objeto, el cual aplica algún algoritmo de extrapolación, no tenga en cuenta las colisiones del mundo y como consecuencia sea capaz de atravesar distintos obstáculos o incluso el propio suelo. Para evitar esto, es necesario aplicar una serie de ajustes. Estos siempre deberán de aplicarse después del cálculo de la extrapolación. Existen muchas formas de abordar este problema, una de ellas es aplicar el motor de físicas en el cálculo de estas posiciones extrapoladas.

Otra técnica de extrapolación es la que posee el DOOM III para predecir futuros estados del resto de jugadores [8]. Para llevarla a cabo, el servidor transmite el *input* más reciente del resto de clientes en cada *snapshot*. Debido a que tanto el cliente como el servidor usan el mismo código para calcular el movimiento del jugador así como las físicas, los *inputs* de la *snapshot* se usan para predecir la siguiente posición del resto de jugadores. En DOOM III el juego recogía *inputs* 60 veces por segundo. Cuando el último *input* se enviaba en la *snapshot* al cliente existía una posibilidad de que el siguiente *input* no fuese idéntico y como consecuencia la predicción fuese errónea. Pero normalmente un jugador no presiona teclas 60 veces por segundo ni cambia su dirección de movimiento de forma tan frecuente. Por lo que en la mayoría de los casos esta técnica daba resultados no muy distantes de la realidad.

Registro de impactos

En los videojuegos multijugador de disparos existen dos tipos de armas [55]. Las más simples son aquellas llamadas armas *hitscan*. En este tipo de armas las balas viajan a una velocidad infinita por lo que en el momento en el que son disparadas se sabe si las balas han colisionado con algo. En contraposición existen otro tipo de armas llamadas armas de proyectil, las cuales poseen un comportamiento mucho más realista puesto que las balas son simuladas de forma física y estas viajan a una velocidad limitada.

En una arquitectura cliente servidor con este último siendo autoritativo, cuando el cliente decide disparar, esta acción se refleja en el comando generado y se empaqueta para ser enviada al servidor. Este será el encargado de realizar los cálculos para detectar si dicha bala acaba colisionando con algún jugador o no. A este proceso se le conoce como *Hit Registration* y en el caso de las armas *hitscan* este se realiza lanzando un *raycast*, una línea recta con el objetivo de buscar los posibles puntos de intersección entre esta con el resto de *colliders*.

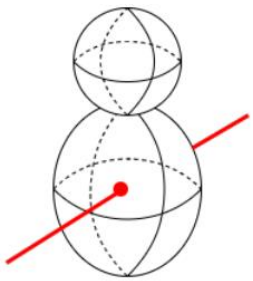


Ilustración 35 - Funcionamiento de un raycast

Esta comunicación entre el cliente y el servidor tiene un problema y está relacionado con la latencia. Cuando el cliente detecta que el jugador ha presionado la tecla encargada de disparar, a pesar de tratarse de un arma *hitscan*, esta no se dispara inmediatamente sino que debe de esperar a que el servidor le confirme si su disparo ha colisionado con un jugador o no. Este latencia generada sería perceptible para el jugador y degradaría la responsividad de la acciones de disparo en el juego. Para solucionar este problema, se podría usar una técnica llamada ocultación de latencia (*Latency Concealment*) [44]. Este método consiste en añadir efectos tanto visuales como sonoros con el objetivo de enmascarar la latencia y hacer que el jugador siga sintiendo el juego responsivo (p.ej. al disparar un arma, el cliente podría mostrar el casquillo de la bala saliendo de la recámara, la animación de retroceso del arma, el sonido

del disparo…) Adicionalmente, también se podría ejecutar el algoritmo de *hit registration* en el cliente para mostrar más efectos en el punto en el que se cree que la bala ha colisionado (p.ej. impactos de balas, sonidos de impacto, sangre…) [55]. Únicamente aquellos efectos que fuesen permanentes (p.ej. reducir la vida del jugador) o que representen la decisión tomada por el servidor como la confirmación del disparo (representada de forma visual en muchos FPS como un *hitmarker*) esperarían a la respuesta por parte del servidor antes de ser ejecutados.

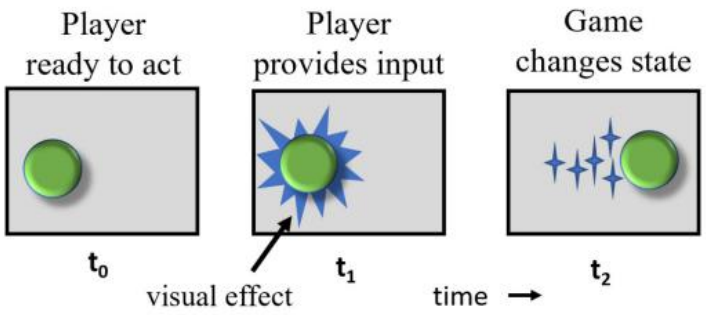


Ilustración 36 - Técnica de ocultación de latencia

Por otra parte, como ya se ha comentado en el apartado de interpolación, el cliente renderiza al resto de jugadores en un estado ligeramente en el pasado al del servidor, concretamente igual a $½\ RTT$ más la latencia generada por el *buffer* de interpolación. Debido a esto podría darse la situación representada en la Ilustración 37 donde en el momento en el que el comando de dicho cliente llegase al servidor la posición del jugador objetivo hubiese cambiado y existiera una probabilidad de que este tomara la decisión de que el disparo ha resultado fallido.

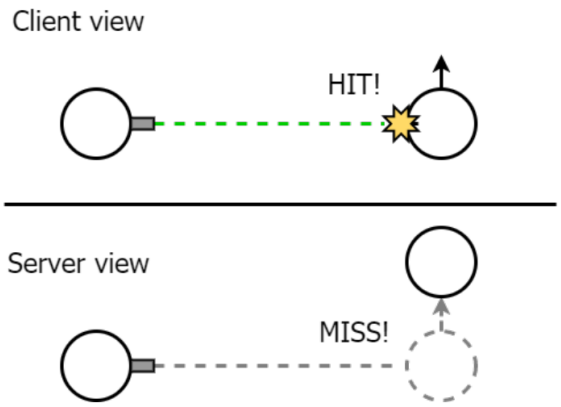


Ilustración 37 - El cliente dispara a un jugador pero cuando el mensaje llega al servidor este ya no se encuentra en dicha posición resultando en un disparo fallido

Para solucionar esto se usa una técnica llamada *Server-side rewind* la cual tiene como objetivo reducir esta desincronización entre el cliente y el servidor replicando en este último el estado de juego de tal forma que este sea idéntico al del cliente en el momento que disparó. Para ello, el servidor necesita almacenar un historial de estados de juego pasados de tal forma que en el momento en el que le llegue una petición de disparo por parte de un cliente, sepa elegir

aquel estado de juego del historial que se corresponde con el momento de la petición. Este proceso puede ser dividido en una serie de pasos a seguir [37]:

1. Calcular el instante de tiempo en el que el cliente realizó la petición de disparo.
2. Elegir el estado de juego pasado del historial que mejor se corresponda al instante calculado en el paso uno.
3. Mover a todos los jugadores a las posiciones en las que se encontraban en el estado de juego elegido del historial.
4. Realizar la técnica del *Hit Registration* para determinar si el disparo resultó exitoso o fallido.
5. Mover a todos los jugadores a sus posiciones actuales para deshacer los cambios del retroceso.

Para calcular el instante de tiempo en el que se realizó la petición de disparo por parte del cliente, es necesario conocer cuánto tiempo atrás es necesario retroceder [55]. Este tiempo será igual al tiempo que ha tardado dicha petición en viajar desde el cliente al servidor, es decir $½\ RTT$ más la latencia que tiene el cliente al renderizar dicha posición en el pasado, es decir $½\ RTT$ más la latencia del *buffer* de interpolación. Con estos datos, la fórmula final quedaría así:

$$InstanteDelDisparo = TiempoActual - RTT - lagBufferInterpolación$$

Ecuación 1 - Fórmula para el cálculo del instante de retroceso de la simulación para compensar la latencia del disparo.

Con este tiempo se llevará a cabo el paso 2 donde el servidor podrá retroceder el estado de juego en el que ocurrió el disparo buscando aquel que mejor se corresponda en el historial de estados de juego pasados. Normalmente el instante de tiempo calculado en el paso 1 será un valor intermedio entre dos estados de juego [37]. En este caso es necesario realizar una interpolación entre los dos estados más cercanos a dicho instante para determinar un estado intermedio.

Esta técnica de retroceso en el servidor será capaz de replicar el estado de juego del cliente siempre y cuando este esté usando técnicas de interpolación [55]. En caso de que el cliente estuviese aplicando la extrapolación en el momento del disparo, y esta haya predicho de forma errónea el estado del jugador objetivo, será muy probable que este error de predicción se traduzca en un disparo fallido.

Una vez encontrado el estado de juego que estaba viendo el cliente en el momento en el que realizó la acción, es necesario mover a todos los jugadores al punto exacto en el que estaban en el momento de la acción. Para ello, el servidor guarda dentro de cada estado de juego del *buffer* las posiciones y rotaciones de todos los *colliders* de cada jugador, los cuales van cambiando a medida que se van simulando las animaciones. Tras haber aplicado a cada jugador sus respectivas posiciones pasadas de los *colliders* se realiza el proceso de *Hit registration* lanzando el *raycast* y comprobando si resultó exitoso o fallido. Tras esto se

vuelven a mover todos los jugadores al estado actual para continuar con el juego. Este método tiene la ventaja de no necesitar realizar un rebobinado de las animaciones evitando así posibles efectos secundarios que puedan surgir. Pero por otra parte, el consumo de memoria es mayor al tener que guardar en cada estado de juego todos los *colliders* de cada jugador.

Otra forma de realizar este retroceso es la que han usado los desarrolladores de Riot Games en Valorant [56]. Al contrario de la anterior implementación, en este caso no se guardan todos los *colliders* de los jugadores en cada estado de juego, sino que estos realizan el retroceso de animaciones. Debido a que esta tarea es bastante costosa decidieron aplicarle una optimización de tal forma que únicamente se retroceden las animaciones de aquellos jugadores que podrían ser alcanzados por un disparo. Para determinar si un jugador se encuentra dentro de la zona de alcance de la bala, el servidor realiza primero un *Sphere cast*, mostrado en la Ilustración 38, desde la posición y con la dirección en la que el jugador desea disparar. Hasta este momento, cada jugador estaba rodeado por un *collider* el cual era el único que se rebobinaba de cara a llevar a cabo este proceso. A la hora de realizar el retroceso de animaciones, debido a que el sistema que usa Valorant es determinista, únicamente guardando los *inputs* relacionados con las animaciones se puede recrear cualquier estado. Por ello en cada estado de juego se guardan estos *inputs* en vez de todos los *colliders* resultantes de las distintas posiciones de los huesos. Antes de aplicar esta optimización actualizar las animaciones de un solo jugador podía llevar hasta 0.1 milisegundos y tras aplicarla todo el procesado de animaciones de un fotograma tardaba una media de 0.3 milisegundos.

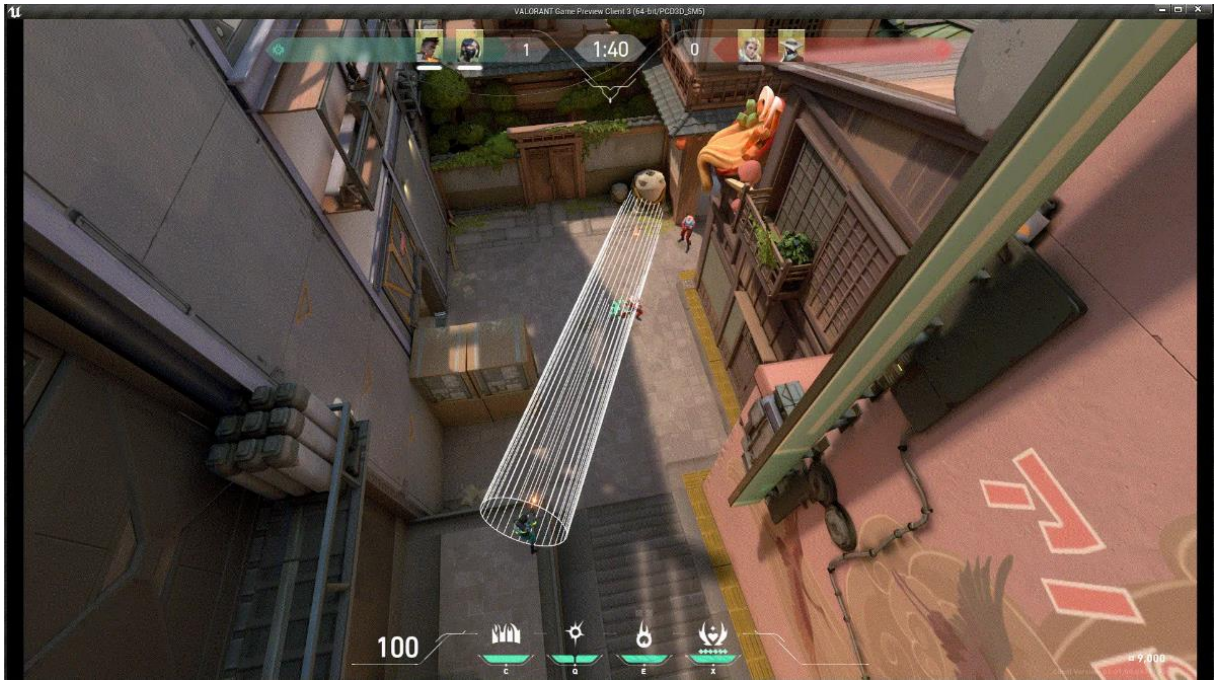


Ilustración 38 - Sphere Cast en Valorant para determinar qué jugadores se encuentran dentro de la zona de alcance de la bala

Otra solución parecida a la desarrollada en Valorant es la que comenta Ford, uno de los desarrolladores de Overwatch, en su ponencia de GDC 2017 [57]. En ella cada entidad (las cuales pueden ser jugadores, puertas, torretas…) posee un *collider* que la contiene. Pero a

diferencia de la solución descrita en Valorant, este *collider* contenedor no solo contiene a la entidad en un instante concreto, sino que abarca todas las posiciones por las que la entidad ha estado comprendidas en un intervalo de tiempo que va desde X milisegundos en el pasado hasta el instante actual. En la Ilustración 39 se puede ver un ejemplo de estos *colliders* en el enemigo situado a la izquierda de la imagen. De esta forma, cuando se realice el *Hit Registration*, primero se comprueba si el *raycast* colisiona con alguno o varios de estos *colliders* contenedores y en caso afirmativo, se rebobinan dichas entidades en base al *ping* del cliente para obtener los resultados finales del disparo.

Ilustración 39 - Collider contenedor de un enemigo en Overwatch el cual no solo contiene la posición actual del jugador sino además una serie de posiciones en el pasado.

Existe otro desafío en las técnicas de compensación de latencia aplicadas a los disparos y es el ser disparado inmediatamente después de haberse puesto a cubierto detrás de un obstáculo *(Shot behind covers*) [22]. Esta situación ocurre cuando un jugador con alta latencia dispara a otro jugador con baja latencia. Es posible que el jugador de baja latencia ya se haya resguardado detrás de un obstáculo y esté fuera del campo de visión del atacante. Sin embargo, debido a la alta latencia del atacante y a que los clientes ven a los enemigos en un estado pasado, el jugador atacado seguirá siendo visible en el estado de juego del atacante. Como resultado, según la Ecuación 1, se calculará una posición en el pasado, lo que podría llevar a la muerte repentina e inesperada del jugador de baja latencia después de haberse cubierto.

Para minimizar este problema, una de las soluciones más usadas es aquella conocida como compensación de lag condicional (*Conditional Lag Compensation*) [58]. Esta técnica hace que el servidor deje de rebobinar los disparos de aquellos jugadores cuya latencia supere un umbral. Los disparos seguirán siendo procesados pero el jugador atacante deberá de apuntar previendo la trayectoria del enemigo para anticiparse a la excesiva latencia. Esta técnica no

solo tiene en cuenta el RTT del cliente atacante, sino también la velocidad de movimiento del jugador debido a que según las fórmulas físicas del movimiento, la distancia recorrida de un cuerpo no solo depende del tiempo sino también de la velocidad. Títulos como Battlefield 4 u Overwatch poseían un límite de 250 milisegundos antes de dejar de compensar mientras que otros como Call Of Duty : Infinite Warfare tenían un límite de 500 milisegundos (excepto en aquellas conexiones P2P donde el atacante era el *host*, las cuales no poseían límite) [59].

Otra solución para disminuir el efecto de los alcances de bala tras haberse cubierto es la que proponen Lee y Chang [60] llamada compensación de lag avanzada (*Advanced lag compensation*) en la que, tras un alcance de bala, la víctima tendrá una oportunidad de denegar este alcance si se detecta una situación de *Shot Behind Covers*. En caso de que una bala haya impactado exitosamente contra un jugador, el servidor enviará a la víctima un evento de haber sido alcanzada por la bala. Este será provisional puesto que dicha víctima tendrá la posibilidad de confirmar o denegar dicho alcance en base a si detecta que ha ocurrido bajo una situación de *Shot Behind Covers*. En caso de confirmarlo, no se realizará ningún cálculo adicional. Por el contrario si se deniega dicha situación, el servidor deberá de verificarlo para evitar que la víctima pueda hacer trampas. Esta técnica añade un retardo adicional a la confirmación del disparo del atacante igual al RTT de la víctima. Esto generaría que para el escenario en el que un jugador con baja latencia disparase a otro jugador con alta latencia el retardo adicional de la confirmación del disparo fuese muy grande. Por ello, los autores han añadido un umbral de latencia máximo. Si la víctima posee una latencia mayor a este umbral el disparo se confirma de forma automática sin tener en cuenta la valoración de la víctima. Esto tiene el objetivo de no empeorar la experiencia de aquellos jugadores con poca latencia por culpa de aquellos con una alta latencia.

Buffer de *jitter*

En los juegos multijugador de ritmo rápido como los FPS, los *inputs* por parte de los clientes deben de aplicarse lo antes posible para así reducir la latencia. Pero, como se ha explicado en el apartado de Desafíos y problemáticas de la red, los paquetes transmitidos a través de la red son vulnerables a llegar tarde a su destino o simplemente no llegar. Debido a que el servidor no puede esperar a que los *inputs* de todos los clientes lleguen a tiempo, en caso de que alguno no haya llegado en el momento de la simulación el servidor realizará una predicción (p.ej. duplicando el *input* más reciente de dicho cliente) y avanzando la simulación [61, 62]. En estos casos existe la posibilidad de que ambos *inputs* (el predicho por el servidor y el del cliente) difieran generando como consecuencia una desincronización. Esto se traducirá en pequeños artefactos visuales en el cliente afectado debido a la reconciliación con el servidor.

Una solución a este problema es introducir el llamado *buffer* de *jitter* o *playout buffer* el cual tiene como objetivo minimizar los problemas causados por el *jitter* a cambio de introducir un poco de latencia. Los *inputs* entrantes en el servidor son almacenados en este *buffer* durante un pequeño intervalo de tiempo antes de ser procesados. Gracias a esto, si un *input* no ha sufrido un retraso excesivo podrá reincorporarse antes de que el servidor realice ninguna operación de predicción sobre este.

Este *buffer* no solo puede aplicarse en el servidor sino que también puede ir en los clientes [63]. Una utilidad del *buffer* de *jitter* en el cliente sería para mejorar la técnica de Interpolación

de entidades en el cliente comentada anteriormente. Y es que, aunque solo se necesiten 2 estados para interpolar, puede pasar que debido al *jitter* o a la pérdida de paquetes los nuevos estados lleguen tarde o simplemente no lleguen. Por ello, es recomendable guardar un historial de varias *snapshots* en un *buffer* para poder estar interpolando de forma constante y garantizar una fluidez visual [37]. El tamaño de este *buffer* puede variar. Si se elige un tamaño relativamente pequeño (p.ej. de una sola *snapshot*), es muy probable que, debido a las problemáticas relacionadas con la red, este se quede vacío muy fácilmente. En contraposición, si su tamaño es demasiado grande, se estaría añadiendo una gran cantidad de latencia artificial constante la cual, sumada a la latencia del cliente con el servidor haría que el resto de jugadores se renderizasen en el cliente en un estado muy anterior al tiempo actual. Esto podría generar problemas en otras técnicas como por ejemplo en el *Hit Registration* [22]. Para calcular la cantidad de latencia añadida [37] se usa la fórmula de $V_l = UT(HS - 1)$ donde UT es el tiempo transcurrido entre dos actualizaciones de *snapshots* consecutivas y HS es el número de *snapshots* del *buffer* cuando la interpolación empieza. Este *buffer* fomenta la consistencia aumentando la viabilidad de los datos en el proceso de interpolación. Como consecuencia, se sacrifica la responsividad generando una constante latencia adicional en el procesado de estos datos.

Algoritmos de gestión de la relevancia

Los algoritmos de gestión de la relevancia son técnicas encargadas de determinar qué información del mundo de juego es relevante para cada jugador [64]. Principalmente, estos algoritmos están basados en la proximidad de un objeto al jugador o en el rango de visión del jugador.

La idea fundamental del funcionamiento de estos algoritmos se basa en el patrón *Observer* [65]. Los publicadores son objetos que producen eventos mientras que los suscriptores son objetos que consumen eventos. Un objeto puede ser un publicador, suscriptor o ambos al mismo tiempo. Cuando un publicador comienza a ser relevante para un suscriptor, este se suscribe a él con la finalidad de ser notificado de los distintos eventos que este produzca. En contraposición, cuando un publicador deje de ser relevante para un suscriptor este se desuscribirá para dejar de recibir futuros eventos.

Por otra parte como se comenta en [64], estos algoritmos pueden estar divididos en (i) aquellos basados en el espacio o (ii) basados en el tipo.

Aquellos basados en el espacio funcionan en base a la proximidad, tomando la posición relativa de los distintos objetos del entorno virtual. Uno de ellos es el modelo espacial de interacción propuesto por Benford y Fahlén [66]. Este modelo usa el espacio como la base de la interacción. A continuación, se van a describir los conceptos clave que conforman este modelo:

1. El medio (*medium*) es un tipo de comunicación por el cual se realiza cualquier tipo de interacción entre los distintos objetos (p.ej. visual, audible…) Cada objeto puede interactuar en uno o varios medios de forma simultánea.

2. El aura es un subespacio, que rodea a un objeto, donde la interacción con otro objeto en un medio específico puede ocurrir.
3. El foco (*focus*) representa el área de interés de un objeto observador en un medio específico.
4. El *nimbus* representa el área en el que un objeto puede ser observado, escuchado, leído, entre otros, por un objeto observador en un medio específico.
5. La conciencia (*awareness*) es el nivel de relevancia que un objeto tiene respecto a otro. Este se calcula a partir del foco y el *nimbus*.
6. Los adaptadores (*adapters*) son objetos capaces de alterar las auras, focos y *nimbus* de otros objetos con el objetivo de modificar su interacción con otros objetos.

A continuación en la Ilustración 40 se muestran varios ejemplos para clarificar los conceptos previos [67]. En un espacio existen dos objetos, un jugador y una televisión, rodeados por líneas discontinuas las cuales delimitan las auras de cada uno de ellos en un medio visual. Cuando ambas auras colisionan entre sí existe una posibilidad de que ambos objetos establezcan una interacción. En este momento, ambos objetos calculan su nivel de conciencia respecto al otro en base al foco del observador (jugador) y al *nimbus* del objeto observado (la televisión). El área de foco del jugador corresponde con su campo visual mientras que el área de *nimbus* de la televisión corresponde con el espacio en el que esta puede ser vista. Por ello en este caso el jugador solo tendrá conciencia de la televisión, y por lo tanto comenzará a recibir sus mensajes, si sus auras han colisionado, la televisión se encuentra dentro de foco del jugador y además el jugador se encuentra dentro del *nimbus* de la televisión.

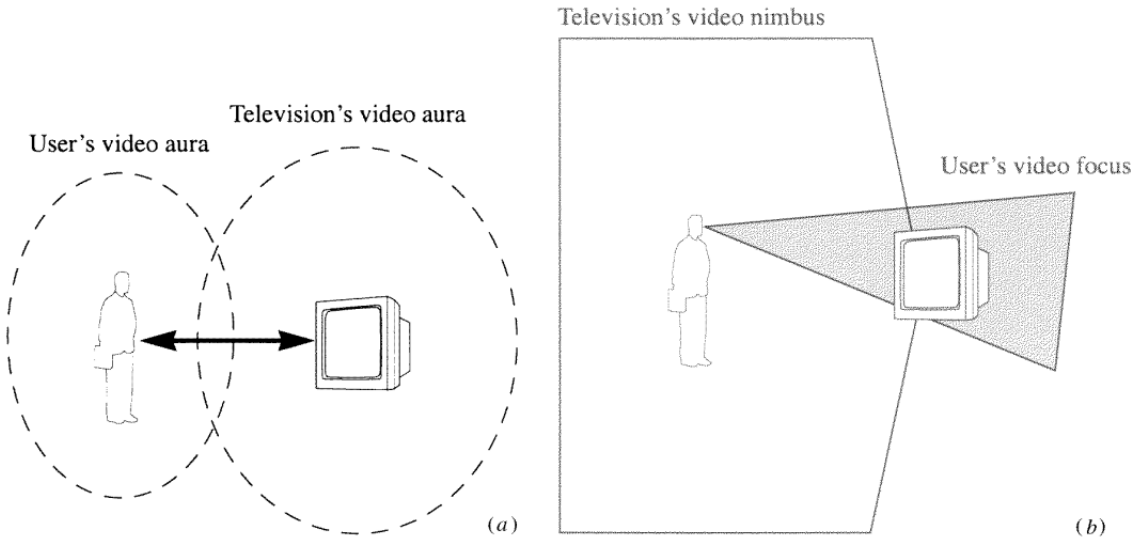


Ilustración 40 - Ejemplo de colisión de auras

Por otra parte, en este modelo el aura, foco y *nimbus* de un objeto pueden ser controlados de varias formas [66]. En primer lugar, por el propio movimiento del objeto. Estos subespacios son capaces de moverse y rotar de igual forma que el objeto al que están asociados. Otra forma puede ser mediante el uso de *inputs* o parámetros. Por ejemplo, el foco de un objeto

puede cambiar como consecuencia de distintos *zoom in/out*. Esto podría ser ocasionado por el jugador al pulsar alguna tecla. Una última forma que proponen Benford y Fahlén es mediante el uso de objetos adaptadores. Por ejemplo, cuando un objeto se acerque a un micrófono, este amplificará tanto su aura como su foco en el medio audible. Como consecuencia un mayor número de objetos podrán recibir estos mensajes.

Este modelo espacial para la gestión de la relevancia permite que los suscriptores únicamente reciban aquellos mensajes que son relevantes para ellos [64]. Pero posee una baja escalabilidad debido a que solo el coste computacional que supone comprobar todas las posibles colisiones entre las auras de los objetos tiene una complejidad de O(MN), donde M es el número de suscriptores y N el número de publicadores en el mundo de juego.

Otro algoritmo dentro de la categoría de algoritmos basados en el espacio es el algoritmo basado en regiones.

3.4 Estudio de alternativas

A la hora de desarrollar un videojuego es necesario elegir tanto el motor de videojuegos como las distintas librerías de software que van a ser usadas en el desarrollo de este. Para ello, es recomendable realizar un estudio de las distintas alternativas que existen en el mercado así como sus ventajas e inconvenientes. Al considerar diversas opciones se elegirá aquella o aquellas que mejor se adapten a las necesidades del proyecto.

En este caso era necesario elegir tanto un motor de videojuegos como una librería multijugador que diese soporte al motor elegido.

Elección del motor de videojuegos

Para la elección del motor de videojuego, se usarán unos criterios de evaluación listados a continuación y ordenados de forma descendente empezando por aquellos de mayor peso:

1. Familiaridad del desarrollador con el software.
2. Calidad de la documentación del software.
3. Soporte multijugador online.
4. Popularidad del software.

A continuación, se explicarán las distintas alternativas evaluadas en la elección del motor de videojuego:

Unreal Engine 5

Unreal Engine es actualmente uno de los motores más usados en la industria de los videojuegos. Es multiplataforma y posee una calidad gráfica excelente y puede ser usado mediante programación en el lenguaje C++ o mediante el uso de una programación visual basada en nodos llamada *Blueprints*.

Unity Engine

Unity Engine es otro de los motores más conocidos y usados en la industria por la cantidad de recursos en línea disponibles tales como cursos, foros o tutoriales. Es multiplataforma y puede ser usado mediante la programación del lenguaje C#.

Godot Engine

Godot Engine es un motor gratuito, de código abierto y multiplataforma. Este puede ser programado usando un lenguaje nativo del motor llamado GDScript aunque en sus últimas versiones se ha incorporado la posibilidad de hacerlo tanto en C# como en C++.

A continuación se muestra la tabla con las alternativas evaluadas para la elección del motor de videojuegos así como los criterios aplicados a cada una de ellas.

Nombre	Familiaridad del desarrollador	Calidad de documentación	Soporte multijugador online	Popularidad
Unreal Engine 5	★☆☆	★★☆	★★☆	★★★
Unity Engine	★★★	★★☆	★★☆	★★★
Godot Engine	★☆☆	★★☆	★☆☆	★★☆

Tras este estudio, se ha decidido optar por **Unity Engine** por ser este el que mejor cumplía los distintos criterios de evaluación establecidos anteriormente.

Elección de la librería multijugador

Para la elección de la librería de red, se usarán los siguientes criterios de evaluación listados de forma descendente empezando por aquel de mayor peso:

1. Familiaridad del desarrollador con el software.
2. Calidad de la documentación del software.
3. Usabilidad y funcionalidades en red del software.

4. Popularidad del software.

A continuación, se mostrarán las distintas librerías multijugador que dan soporte a **Unity Engine**.

Netcode For Gameobjects

Netcode For Gameobjects es una librería de red de alto nivel multiplataforma para **Unity Engine** la cual permite al programador abstraerse de los detalles de bajo nivel como por ejemplo los protocolos. Es una de las más populares dentro del entorno de **Unity Engine** y de las que más tiempo llevan operativas siendo esta una sucesore de la antigua MLAPI.

Photon

Photon posee un conjunto de librerías de alto nivel multiplataforma para la creación de videojuegos multijugador. Cada una de ellas posee una serie de características. **Photon** ofrece muchas funcionalidades tales como servicios de *Matchmaking*, chat de voz o técnicas de compensación de latencia ya implementadas.

Coherence

Coherence es un nuevo motor de red el cual permite crear videojuegos multijugador en muy poco tiempo. Posee una gran cantidad de herramientas para poder crear el multijugador sin apenas escribir código así como algunas funcionalidades ya implementadas para compensar la latencia.

Nombre	Familiaridad del desarrollador	Calidad de la documentación	Usabilidad y funcionalidades en red	Popularidad
Netcode For Gameobjects	★★★	★★★	★★★	★★★
Photon	★☆☆	★★★	★★★	★★★
Coherence	★☆☆	★★★	★★★	★★☆

Tras este estudio, se ha decidido usar **Netcode For Gameobjects** por ser este el que mejor se adapta a los criterios de evaluación fijados.

Capítulo 4

Descripción informática

En esta sección se van a detallar los distintos requisitos técnicos de este proyecto así como un análisis minucioso y detallado de la implementación de la arquitectura cliente-servidor o de los distintos algoritmos y técnicas multijugador implementadas en el código fuente. Asímismo, esta sección proporciona una visión profunda de la complejidad técnica de dichas soluciones multijugador.

5.1 Análisis

Requisitos funcionales y no funcionales

A continuación se pasarán a listar tanto los requisitos funcionales como los no funcionales de este proyecto. Cada requisito tendrá un código asociado, el cual empezará por "RF" para los requisitos funcionales y por "RNF" para los requisitos no funcionales, y una breve descripción.

Código	Descripción del requisito
RF01	Los clientes deben poder enviar sus *inputs* al servidor.
RF02	Los clientes deben de poder recibir las *snapshots* enviadas por el servidor.
RF03	El cliente debe ser capaz de predecir el estado futuro del jugador anticipándose los resultados del servidor.
RF04	El servidor debe de ser capaz de compensar la latencia en el cálculo de las operaciones de disparos del resto de clientes.
RF05	El cliente debe de ser capaz de interpolar los distintos estados de las entidades en red que lleguen por parte del servidor.
RF06	El cliente debe de ser capaz de predecir futuros estados de las entidades en red en caso de que las técnicas de interpolación no puedan llevarse a cabo.

RF07	El servidor debe de ser capaz de predecir aquellos *inputs* que lleguen demasiado tarde o simplemente no lleguen.
RF08	El cliente debe de ser capaz de adoptar el estado del servidor tras una reconciliación.
RF09	El cliente debe de ser capaz de detectar cuando se ha interrumpido la conexión con el servidor y saber gestionarlo.
RF10	El juego debe de ser capaz de gestionar a través de la red tanto entidades controladas por otros clientes (jugadores) como entidades autocontroladas o controladas por el servidor.
RF11	El cliente debe de ser capaz de conectarse a un servidor mediante el uso de una conexión *loopback* así como mediante el uso de una conexión a internet con un *Servidor de relay* intermedio.

Código	Descripción del requisito
RNF01	La autoridad de las decisiones tomadas en el sistema debe de recaer única y exclusivamente en el servidor.
RNF02	El prototipo debe de minimizar u ocultar el retraso generado por la latencia de la red para maximizar la responsividad de las acciones de los jugadores.
RNF03	El proceso de avance de la simulación debe de ser determinista.
RNF04	El cliente debe de ser capaz de recuperarse tras una desincronización con el servidor.
RNF05	La probabilidad de que aquellos clientes con malas conexiones afecten a los clientes con buenas conexiones debe de ser baja.
RNF06	Los clientes deben de ser capaces de ocultar los efectos de la latencia en conexiones con RTT menores a 100 milisegundos.

Especificaciones

A continuación, se van a listar las diferentes versiones de todos los programas, librerías y otras herramientas usadas en la creación de este prototipo FPS multijugador en línea:

Nombre de la herramienta	Uso de la herramienta	Versión
Unity Engine	Motor del prototipo	2021.3.26f1

Netcode For GameObjects	Solución de soporte multijugador a bajo nivel para Unity Engine.	1.4.0
Visual studio	Entorno de desarrollo integrado (IDE por sus siglas en inglés) para la programación del prototipo.	Community 2022 17.6.3
Cinemachine (Virtual Camera)	Solución desarrollada por Unity Engine para la creación y gestión de cámaras.	2.8.9
Clumsy	*Software* para la simulación de condiciones de la red artificiales.	0.3

5.2 Diseño

Arquitectura de los componentes en red claves del proyecto

El diseño de la arquitectura de *netcode* de este prototipo toma como referencia principalmente las arquitecturas de Quake III Arena y DOOM III. En la estructura de este prototipo, existe un componente centralizado encargado de gestionar la mayor parte del *netcode* de alto nivel llamado `NetworkController`. Este componente es compartido tanto en el modo cliente como en el modo servidor. No obstante, el componente centralizado hace uso de elementos específicos en base al tipo de host con el que se configure. Por ejemplo, en el modo cliente, existe un componente encargado de decodificar y distribuir la información de una *snapshot* a las distintas entidades interesadas, mientras que en servidor, existe un componente capaz de llevar a cabo el algoritmo de *Hit Registration* cuando sea necesario. Esto significa que este componente es capaz de adaptarse y configurarse en base al tipo de *host* que se habilite, con la finalidad de atender a las necesidades específicas de cada modo.

En la Ilustración 41 se muestra un ejemplo gráfico de cómo se personaliza la inicialización de este componente centralizado, llamado `NetworkController`, en base al tipo de *host*. En esta ilustración se puede apreciar que el servidor hace uso de muchos más componentes que el cliente. Esto en parte es debido al concepto de *Dumb client*. El cliente se centra únicamente en establecer la conexión con el servidor y en decodificar y procesar las distintas *snapshots* que recibe. Por otro lado, el servidor, el cual posee plena autoridad sobre el estado del juego, se encarga de todas las tareas relacionadas con este estado de juego, entre las que se pueden destacar el avance de la simulación del mundo virtual o el procesamiento de las distintas peticiones de disparo de los clientes mediante un algoritmo de *Hit Registration*.

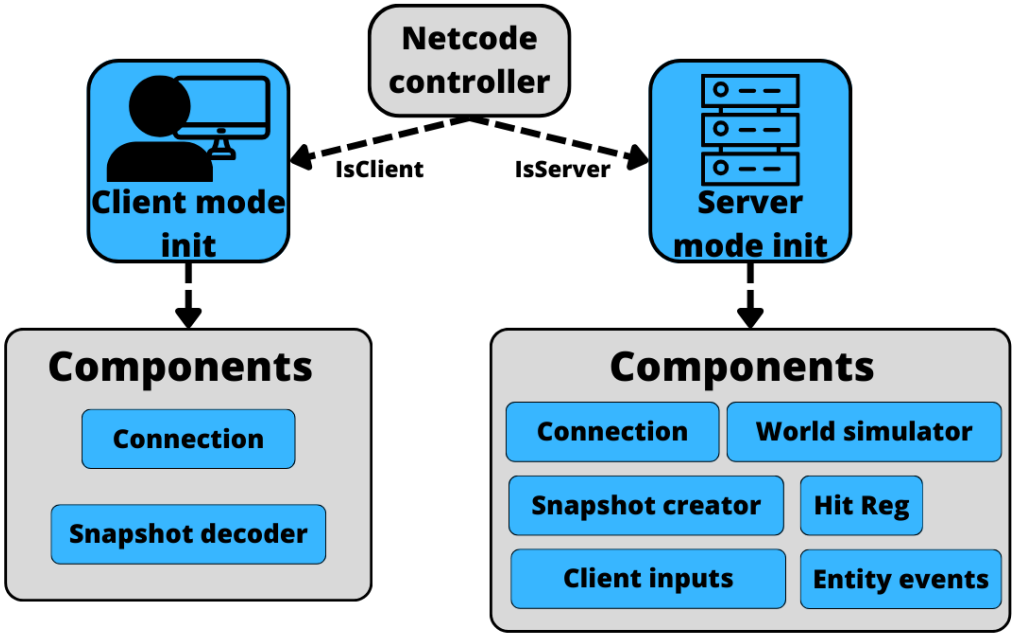


Ilustración 41 - Inicialización personalizada del componente principal de netcode

Otro aspecto clave de la estructura de esta arquitectura reside en las distintas entidades sincronizadas a través de la red. Estas entidades mantienen una comunicación bidireccional con el `NetworkController`. Cada entidad guarda una referencia al `NetworkController` para realizar una sola petición: El envío de inputs al servidor.

A su vez, el `NetworkController` se comunica con las distintas entidades mediante dos interfaces:

1. `INetworkEntity` en caso de ser una entidad no controlada por los *inputs* de un jugador (p.ej. Entidades temporales, *bots*…)

2. `IPlayerNetworkEntity` en caso de ser una entidad controlada por los *inputs* de un jugador.

La comunicación en esta dirección es mucho más exhaustiva, ya que el componente central de *netcode* delega muchas funciones en las propias entidades. Por ejemplo, estas funciones pueden implicar el avance de la simulación de una entidad concreta, el retroceso de su estado u la obtención de cierta información relevante como el identificador único de dicha entidad o del cliente que la controla.

Gracias a esta comunicación por parte del NetworkController con las distintas entidades mediante el uso de interfaces, permite la creación de múltiples tipos de implementaciones concretas de entidades sin necesidad de modificar el componente central de *netcode*. Como

consecuencia, la implementación de nuevas funcionalidades en red dentro del prototipo se realiza de forma más rápida.

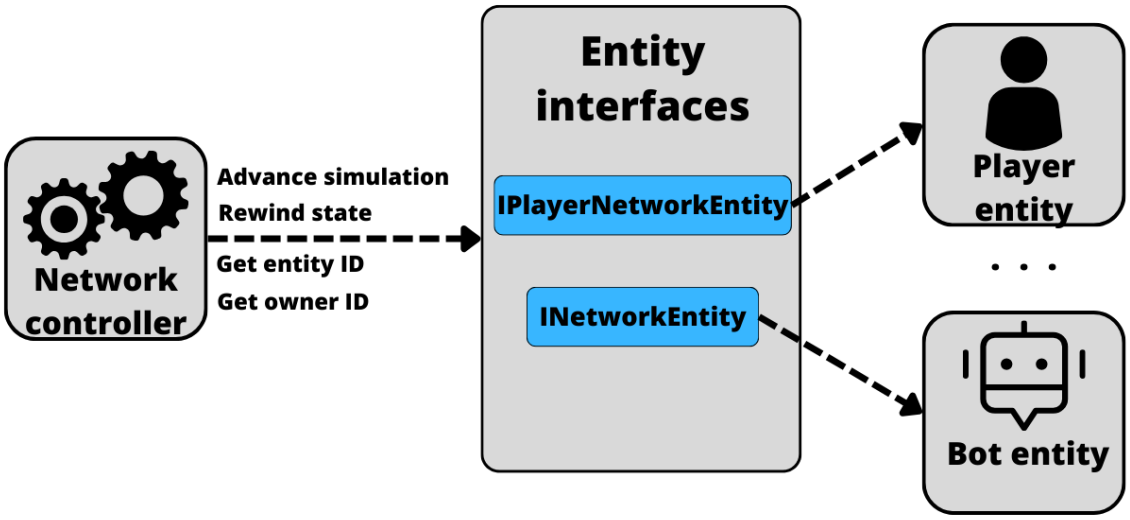


Ilustración 42 - Comunicación del NetworkController con las distintas interfaces de entidad

Comunicación en red entre cliente y servidor

La comunicación que un cliente y un servidor mantienen durante el transcurso de una sesión de juego es muy simple. El cliente genera *inputs* a una frecuencia de 50Hz los cuales son enviados al servidor. A su vez, el servidor, tras cada proceso de simulación del mundo del juego, empaqueta los resultados en una *snapshot* para enviarla de forma personalizada a cada uno de los clientes conectados.

Llegada de un input al servidor

Cuando el servidor recibe un *input* por parte de un cliente conectado, este no lo procesa de forma inmediata, sino que lo almacena en un *buffer* a la espera de ser procesado. Aunque se proporcionarán más detalles en el apartado de Desarrollo, una de las razones para almacenar estos *inputs* en el servidor es para mitigar los efectos del posible *jitter* que esas conexiones puedan tener.

El *buffer* garantiza que en cada *tick* de la simulación, el servidor procese solo un *input*, salvo en excepciones. De esta forma, si por alguna razón, el servidor recibe 2 *inputs* consecutivos de forma simultánea por parte de un mismo cliente, en caso de ser válidos, se almacenarán en el *buffer* y se irán retirando uno a uno a medida que el estado del juego avance.

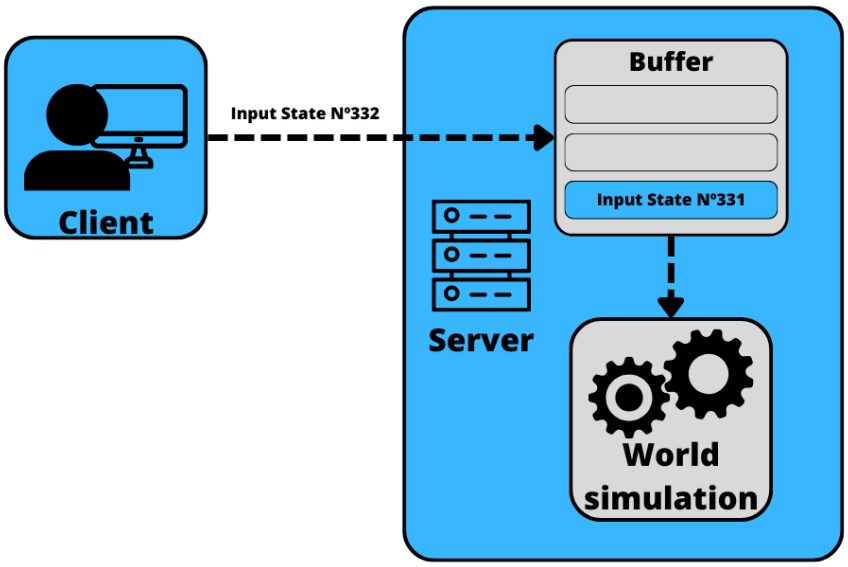


Ilustración 43 - Llegada de un input al servidor

Llegada de una snapshot al cliente

Cuando el cliente recibe una *snapshot* por parte del servidor este lleva a cabo un proceso de decodificación con el fin de distribuir su contenido entre las distintas entidades interesadas. Este proceso trata de detectar aquellas entidades cuyos identificadores se encuentran dentro de la *snapshot*. Una vez identificadas, se envía la información asociada a cada una de ellas de dentro de la *snapshot*. Este envío de información también se realiza a través de las interfaces `INetworkEntity` e `IPlayerNetworkEntity` con la finalidad de abstraer el componente centralizado de *netcode*, el cual es el responsable de llevar a cabo este procesamiento de *snapshots* en el cliente, del resto de entidades.

Una vez la información del nuevo estado proveniente de una *snapshot* es recibido por la entidad a la cual está asociado, será esta la responsable de realizar las operaciones necesarias para actualizar su estado actual en base a la información recibida.

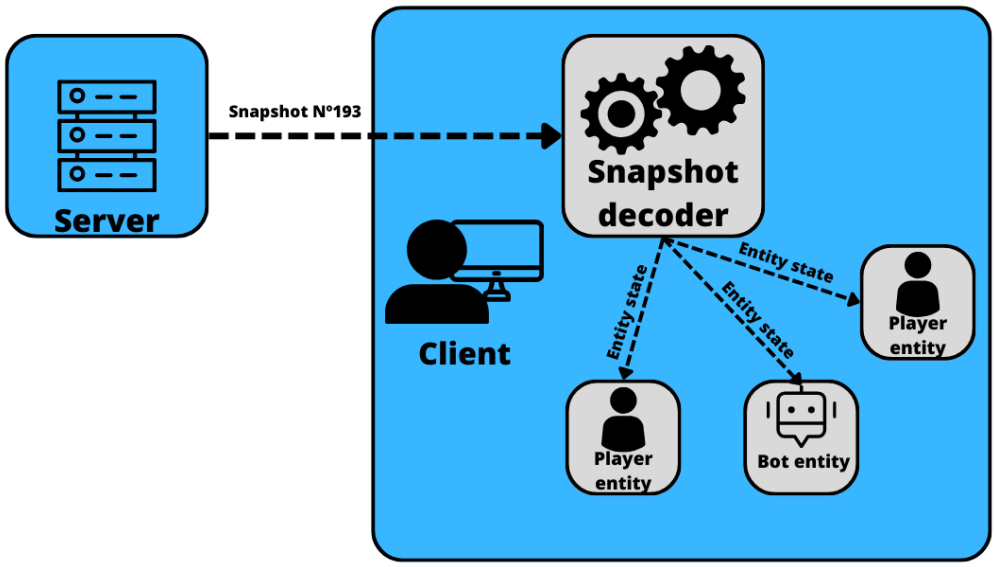


Ilustración 44 - Llegada de una snapshot en el cliente

En el lado del cliente, cuando una entidad recibe un nuevo estado procedente de una *snapshot*, las operaciones realizadas dependen del tipo de entidad que esté recibiendo la información. En relación con este aspecto, una entidad puede ser de tres tipos:

1. Entidad predicha: Este tipo de entidad es aquella controlada por el jugador local, y emplea un sistema de predicción en el lado del cliente así com un Sistema de reconciliación con el servidor. En este caso, la información recibida por parte de la *snapshot* se utiliza para comprobar si los resultados de la simulación en el cliente corresponden con aquellos recibidos del servidor. Si hay discrepancias, se lleva a cabo una reconciliación y posteriormente se resimulan todos los *inputs* posteriores al estado recibido de la *snapshot.*
2. Entidad interpolada: Estas entidades no son controladas por el jugador local, sino por otro jugador o incluso el propio servidor (p.ej. un *bot*). Aquí, la entidad usa un sistema de interpolación y extrapolación de estados. En este caso, la información recibida por parte de la *snapshot* se envia al componente de interpolación para su procesamiento posterior.
3. Entidad temporal: Son entidades especiales cuyo único propósito es transmitir eventos del servidor al cliente. En este caso, la información recibida por parte de la *snapshot* se utiliza para configurar y ejecutar un evento de entidad en el juego (p.ej. un hitmarker o un impacto de bala, entre otros)

5.3 Desarrollo

En este apartado se detallará cómo se han estructurado los diversos componentes de código encargados de llevar a cabo las funcionalidades multijugador del prototipo FPS así como su funcionamiento interno o las distintas relaciones que existen entre ellos. Además, en este apartado se mostrarán distintos diagramas de flujo para representar de forma gráfica los distintos procedimientos que se siguen tanto en el cliente como en el servidor. Estos diagramas de flujo poseen un fondo de color salmón para aquellos que representen un conjunto de tareas ejecutadas en el servidor y un fondo de color azul claro para aquellos que representen un conjunto de tareas ejecutadas en un cliente.

Estructuras de datos principales

En primer lugar, se van a introducir las distintas estructuras de datos que se transmiten a través de la red para llevar a cabo la comunicación entre cliente y servidor. Estas se encuentran ilustradas en la Ilustración 45.

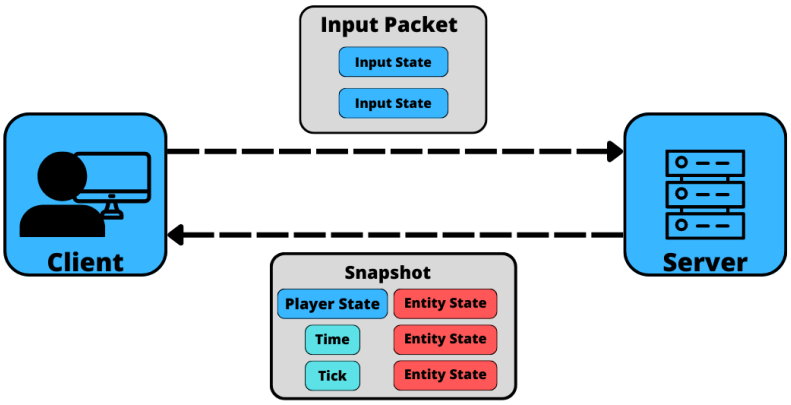


Ilustración 45- Estructuras de datos transmitidas a través de la red en la comunicación entre los clientes y el servidor

Input packet e Input states

La única información que transmite el cliente hacia el servidor son los *inputs* generados para su procesamiento. Esta información se almacena en una estructura denominada `Input Packet`, la cual contiene tanto el *input* actual del cliente como los X *inputs* inmediatamente anteriores. La decisión de almacenar los *inputs* pasados se ha tomado con el objetivo de reducir las posibilidades de que un *input* no llegue al servidor, y se explicará con mayor detalle en la sección de Obtención de inputs. Cada *input* dentro del `Input Packet` se representa mediante otra estructura de datos llamada `Input State`, que incluye las teclas presionadas por el jugador en un momento específico, así como el *tick* interno del cliente. Este *tick* se utiliza exclusivamente como un identificador del *input* para poder realizar comparaciones entre los estados del jugador local, predichos por el cliente, y los estados que lleguen desde el servidor, con el fin de determinar si es necesario llevar a cabo una reconciliación. Esto se explicará con

mayor detalle en el apartado de Proceso de predicción en el lado del cliente de estados futuros del jugador local. Gracias a esta estructura se cumple el requisito RF01

Snapshot state

Por otra parte, el servidor envía a todos los clientes una estructura de datos de mayor tamaño llamada `Snapshot State`. Esta *snapshot* está personalizada para cada uno de los clientes a los que se le manda y contiene los siguientes datos:

1. Un booleano indicando si el cliente destinatario ha generado algún estado de su jugador local en el *tick* actual.
2. Una estructura de tipo `Player State` la cual posee toda la información del estado del jugador local de dicho cliente.
3. Un booleano indicando si la *snapshot* contiene estados para otras entidades.
4. Un *array* con el conjunto de entidades en red de la simulación.
5. El instante de tiempo del servidor en el que se generó dicha *snapshot*.
6. El *tick* del servidor al que la información de la *snapshot* pertenece.

Player state

Uno de los elementos guardados en la *snapshot* es una estructura llamada `Player State`. Esta contiene toda la información necesaria para representar el estado de un jugador y contiene los siguientes datos:

1. La posición y rotación del jugador.
2. Información sobre los distintos sistemas que moldean el comportamiento del jugador (p.e. Sistema de movimiento, Sistema de retroceso del arma, Sistema de apuntado del arma…)
3. El identificador del objeto en red de dicho jugador.
4. El *tick* interno del último *input* que ha simulado.

Entity state

El `Entity State` es una estructura de datos encargada de representar el estado de una entidad haciendo una criba descartando toda aquella información que no es necesaria para un cliente. Por ejemplo, para representar un jugador enemigo en un cliente no es necesario enviar toda la información contenida en la estructura de `Player State`, sino que únicamente con la posición, rotación y algunas variables para las animaciones sería suficiente. Esta información es la necesaria para poder llevar a cabo los algoritmos de interpolación de entidades de los clientes. A continuación, se enumerarán los distintos datos que contiene esta estructura de datos:

1. La posición y rotación de la entidad.
2. El tiempo del servidor en el que se generó dicho estado.
3. El identificador del objeto en red de dicha entidad.

4. Algunas variables relacionadas con las animaciones.
5. Un evento de entidad.

Visión general del conjunto de sistemas que actúan en la comunicación entre cliente y servidor

A continuación se va a pasar a detallar el proceso desde que se genera un *input* en el cliente hasta que la *snapshot* asociada a dicho *input* es recibida y procesada en el mismo cliente. Este proceso se puede mostrar de forma gráfica en el diagrama secuencia de la Ilustración 46.

En primer lugar, el cliente crea una estructura de `Input Packet` con el *input* actual y el *input* inmediatamente anterior. Tras esto, el cliente lo envía a través de una llamada RPC al servidor para su procesamiento. Además, el cliente realiza la predicción del estado futuro del jugador local y lo guarda en el *buffer* de estados. Por último, el cliente comprueba si han llegado nuevos estados por parte del servidor para realizar una comparación y determinar si es necesario llevar a cabo una reconciliación y como consecuencia una resimulación para corregir los estados posteriores al estado con el que se ha reconciliado el cliente.

Cuando el `Input Packet` llega al servidor, este lo almacena en un *buffer* de *jitter* de *inputs* a la espera de ser procesado. El tiempo que un *input* permanecerá en este *buffer* varía entre 0 y un máximo de 3 *ticks*. Una vez ha transcurrido este tiempo, el *input* será sacado del *buffer* y será usado para avanzar la simulación del jugador local de dicho cliente. Al finalizar el *tick*, el servidor empaquetará los resultados en una *snapshot* y la enviará al cliente.

Una vez la *snapshot* llegue al cliente, este se encargará de repartir la información entre las distintas entidades. Primero enviará la información al jugador local quien guardará el estado recibido por parte del servidor para ser comparado en el siguiente *tick*. Posteriormente se enviará uno por uno el estado de entidad a las distintas entidades. En caso de que las entidades sean interpoladas (p.ej. los jugadores remotos) el estado de la entidad se guardará en un *buffer* de *jitter* para su posterior uso en los algoritmos de interpolación o extrapolación. En caso de que no sea una entidad interpolada (p.ej. las entidades temporales) se llevará a cabo un procesamiento de dicho estado según se haya programado en cada entidad.

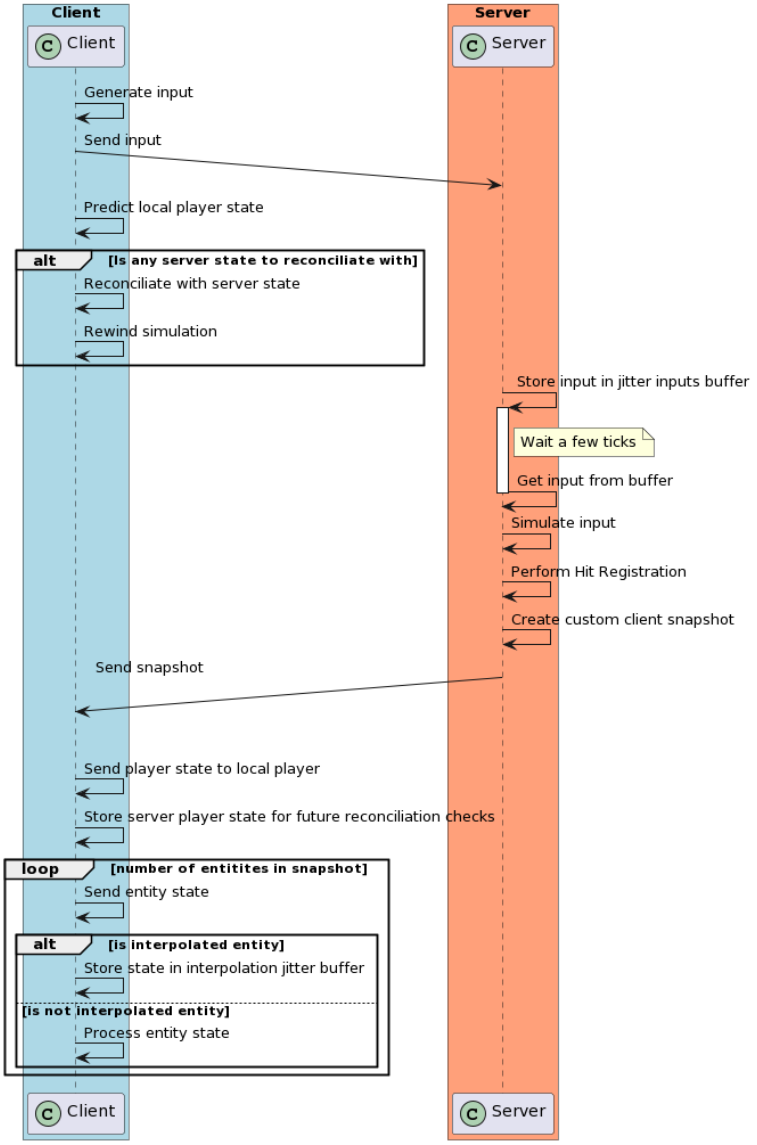


Ilustración 46 - Flujo de tareas entre cliente y servidor desde que se genera un input hasta que se recibe su respuesta en el cliente

Controlador de *netcode* centralizado

Toda la gestión del código encargado de realizar las distintas operaciones relacionadas con el *netcode* tanto en el cliente como en el servidor se encuentra centralizada en un único controlador. A continuación se pasará a detallar las distintas tareas de las que se encarga este controlador.

Inicialización personalizada

La inicialización de este controlador es distinta tanto para un cliente como para un servidor. En este proceso, el controlador valorará qué tipo de nodo es la máquina y en base a eso inicializará unos sistemas u otros. Por ejemplo, no tiene sentido que un cliente inicialice el sistema encargado de generar *snapshots* puesto que esta tarea es única y exclusiva del servidor. Esto no solo ahorra memoria sino que de esta forma los desarrolladores pueden

detectar *bugs* y comportamientos erróneos en caso de que un cliente intente acceder a un sistema exclusivo de un servidor o viceversa.

Como se puede apreciar en la Ilustración 47 e Ilustración 48 el `NetworkController` del servidor posee una mayor cantidad de sistemas que en el cliente. Esto hace referencia al hecho de que los clientes sean máquinas "tontas" (*Dumb Clients*) y la gran mayoría de tareas sean representar la información recibida por parte del servidor.

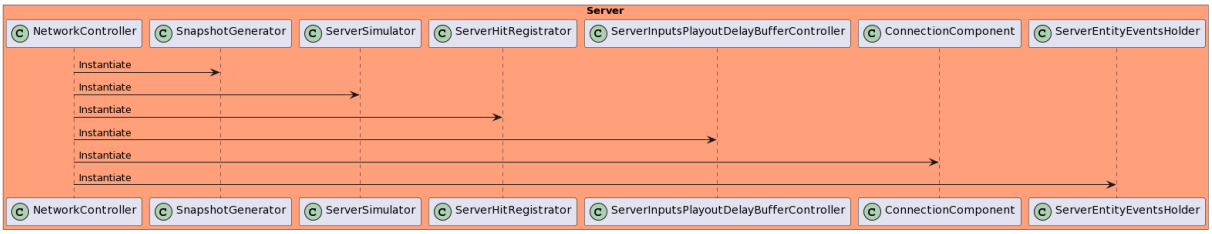


Ilustración 47 - Diagrama secuencia de la inicialización de los distintos sistemas del NetworkController en el servidor

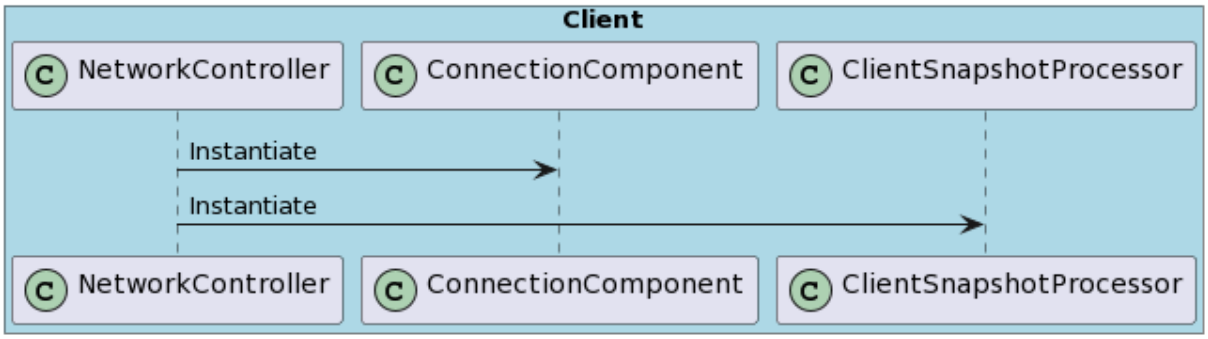


Ilustración 48 - Diagrama secuencia de la inicialización de los distintos sistemas del NetworkController en el cliente

Además, en caso de que el `NetworkController` sea creado en una instancia de servidor este desactivará todos los componentes de renderizado de todos los objetos en escena por temas de optimización de cómputo.

Punto de acceso centralizado

El controlador `NetworkController` es el único que posee referencia a estos sistemas. Ningún otro script ajeno a él puede obtener una referencia a dichos sistemas y por lo tanto, acceso a los mismos. Esto se ha implementado como medida de seguridad. El controlador ofrece distintos métodos públicos para realizar ciertas tareas que involucren la manipulación de uno o varios de los sistemas del `NetworkController`. Por esta razón, se podría decir que este controlador hace uso del patrón Fachada [68] puesto que hace de intermediario entre los consumidores y los sistemas del `NetworkController`. Un ejemplo podría ser el método `SpawnTemporaryEntity` el cual, instancia un entidad temporal asociándola un evento de entidad. Este método hace uso de los sistemas de control de eventos de entidad (`ServerEntityEventsHolder.cs`) y de control de las entidades activas en la simulación (`ServerSimulator.cs`). De esta forma, el consumidor únicamente se encarga de proporcionar la información necesaria al `NetworkController` y este ya se encarga de

gestionar y realizar todo el proceso. Además, como se ha comentado en el apartado de Inicialización personalizada, este método lanzará un error en caso de que se ejecute desde un cliente puesto que únicamente el servidor debería de poder instanciar entidades temporales.

Llegada de un *input* al servidor

Como se ha mencionado en el apartado anterior, cada cliente envía mensajes de tipo `InputPacket` al servidor para que estos sean procesados. Cuando el servidor recibe un `InputPacket` por parte de un cliente, este lo procesa y almacena aquellos *inputs* que sean válidos en un Buffer de jitter.

Tipos de entidades gestionadas en la simulación

El sistema encargado de realizar el cálculo de la simulación y avanzar el estado de juego diferencia dos tipos de entidades.

Entidades por defecto

En primer lugar están las entidades por defecto que son todas aquellas que no necesitan de un parámetro de tipo `InputState` para avanzar su estado en el juego. Un ejemplo de este tipo de entidades son los muñecos diana implementados en el nivel. Estas entidades implementan la interfaz `INetworkEntity` la cual posee los siguientes métodos:

1. `IsRewindable`: Indica si la entidad tiene la capacidad de cambiar su estado a lo largo del tiempo y por lo tanto ser retrocedida para realizar técnicas de compensación de la latencia.
2. `Rewind`: En caso de que `IsRewindable` sea verdadero (*true*), este método retrocederá la entidad a un instante de tiempo pasado por parámetro.
3. `IsPlayer`: Indica si esta entidad es de tipo `IPlayerNetworkEntity`.
4. `GetNetworkEntityId`: Devuelve el identificador de esta entidad.
5. `GetClientOwnerId`: Devuelve el identificador del cliente que posee esta entidad.
6. `ReceiveEntityState`: Recibe un `EntityState` para ser procesado por la entidad (p.e. para realizar una interpolación del estado de la entidad).
7. `GetEntityStateWithoutEvent`: Devuelve el estado actual de la entidad sin los datos asociados a los eventos de entidad.
8. `Simulate`: Avanza el estado de la entidad en base a un parámetro de tiempo transcurrido.
9. `GetNetworkObject`: Devuelve el `NetworkObject` asociado a esta entidad.

Entidades jugador

Por otra parte, el otro tipo de entidad son las entidades jugador, las cuales son aquellas que necesitan de un parámetro de tipo `InputState` para avanzar su estado de juego. Un ejemplo de este tipo de entidad son todos aquellos jugadores controlados por los distintos clientes de la partida. Las entidades jugador implementan la interfaz `IPlayerNetworkEntity` la cual, aparte de implementar todos los métodos de la interfaz `INetworkEntity`, implementa los siguientes métodos adicionales:

1. `Simulate`: Este método avanza el estado de la entidad en base a un `InputState` pasado por parámetro.
2. `GetPlayerState`: Devuelve el estado actual de la entidad jugador.
3. `ReceivePlayerState`: Recibe un estado de la entidad jugador para ser procesado (p.e. realizar una reconciliación si este estado diverge).

Cuando una entidad es instanciada en el servidor, este deberá de añadirla al conjunto de entidades de la simulación para que pueda ser tenida en cuenta por los distintos algoritmos que hacen uso de estos datos. De igual forma, cuando esta entidad deje de existir en la simulación, esta deberá de ser borrada para evitar errores de referencias nulas.

Eventos de entidad y entidades temporales

En el apartado de Estructuras de datos principales se han descrito las estructuras más relevantes que el código de *netcode* hace uso en este prototipo. Entre ellas se encontraba la estructura de `Entity State` la cual posee un parámetro llamado `EntityEvent`. Al igual que en muchos videojuegos multijugador en línea como el Quake III Arena, en este prototipo cada entidad tiene la capacidad de invocar un evento en cada *snapshot*. Estos eventos representarán acciones llevadas a cabo por una entidad en el servidor que tendrán una serie de consecuencias en el cliente. Por ejemplo, si una entidad decide disparar su arma se creará un evento asociado a esa entidad el cual cuando llegue al cliente desembocará en una serie de efectos visuales y sonoros (Sonido de disparo, animación de disparo para dicha entidad…). Estos eventos tienen dos campos principales, (i) un identificador de evento y (ii) un *array* de *bytes* reservado para almacenar parámetros. Este último campo es opcional puesto que dependerá del tipo de evento. Volviendo al ejemplo anterior, para el evento invocado como consecuencia del disparo del arma de una entidad, será necesario almacenar como parámetro el identificador del arma con el que se disparó. De esta forma, será posible configurar el tipo de sonido que desencadene en el cliente.

Por otra parte, existen ciertos eventos los cuales no están directamente relacionados con ninguna entidad. Por ejemplo, si se tiene un evento el cual es invocado cada vez que una bala impacta contra una superficie, dicho evento no estaría relacionado con ninguna entidad. Debido a esto, existe un tipo de entidad llamada entidad temporal cuyo único propósito es transmitir el evento a los distintos clientes. Una vez cumplido su cometido esta se volverá innecesaria y será destruida (de ahí su nombre de “temporal”). Sin embargo, estas entidades temporales también pueden crearse para aquellos eventos con un gran número de parámetros. Al instanciar una entidad temporal no solo se tienen los bytes del *array* del parámetros del evento de entidad sino también los campos de posición y rotación de dicha entidad temporal.

A continuación, se va a pasar a detallar los distintos eventos de entidad que existen para el caso de este prototipo FPS multijugador en línea:

1. `EV_FIRE_WEAPON`:
    a. Descripción: Este evento es invocado cada vez que una entidad dispara su arma.

   b. Parámetros:
      i. ID del arma.
   c. Usado en entidad temporal: No.
   d. Consecuencias en el cliente:
      i. Animación de disparo.
      ii. Sonido de disparo personalizado en base al ID del arma.
2. `EV_RELOAD_WEAPON`:
   a. Descripción: Este evento es invocado cada vez que una entidad recarga su arma.
   b. Parámetros:
      i. ID del arma.
   c. Usado en entidad temporal: No.
   d. Consecuencias en el cliente:
      i. Animación de recargar.
      ii. Sonido de recargar personalizado en base al ID del arma.
3. `EV_BULLET_HIT_NON_SHOOTABLE`:
   a. Descripción: Este evento es invocado cada vez que una bala colisiona contra un objeto que no esté en la capa Shootable.
   b. Parámetros:
      i. Vector posición del impacto.
      ii. Vector normal de la superficie impactada.
      iii. ID del tipo de material de dicha superficie (Metal, Madera…)
   c. Usado en entidad temporal: Sí.
   d. Consecuencias en el cliente:
      i. Instanciación de efecto de partículas personalizado según el ID del material en el vector posición mirando hacia el vector normal.
      ii. Instanciación de un *sprite* mostrando un agujero de bala personalizado según el ID del material en el vector posición mirando hacia el vector normal.
4. `EV_BULLET_HIT_SHOOTABLE`:
   a. Descripción: Este evento es invocado cada vez que una bala colisiona contra un objeto que esté en la capa Shootable.
   b. Parámetros:
      i. Vector posición del impacto.
      ii. Vector normal de la superficie impactada.
      iii. ID del tipo de material de dicha superficie (Metal, Madera…)
      iv. ID de la entidad que efectuó el disparo.
      v. ID de la entidad que recibió el disparo.

c. Usado en entidad temporal: Sí.
d. Consecuencias en el cliente:
    i. Instanciación de efecto de partículas personalizado según el ID del material en el vector posición mirando hacia el vector normal.
    ii. Instanciación de un *sprite* mostrando un agujero de bala personalizado según el ID del material en el vector posición mirando hacia el vector normal.
    iii. Si el ID de la entidad atacante es igual al ID del jugador local de dicho cliente activar el *Hitmarker*.
    iv. Si el ID de la entidad víctima es igual al ID del jugador local de dicho cliente activar un indicador de daño con apuntando en la dirección de la normal de la superficie.

Etapas de un *tick* en el servidor

En cada *tick*, el servidor realiza una serie de operaciones con el fin de actualizar el estado de juego a partir de los nuevos *inputs* recibidos por parte de los clientes y transmitir a estos los resultados. Este proceso se divide en cuatro tareas fundamentales representadas gráficamente en la Ilustración 49.

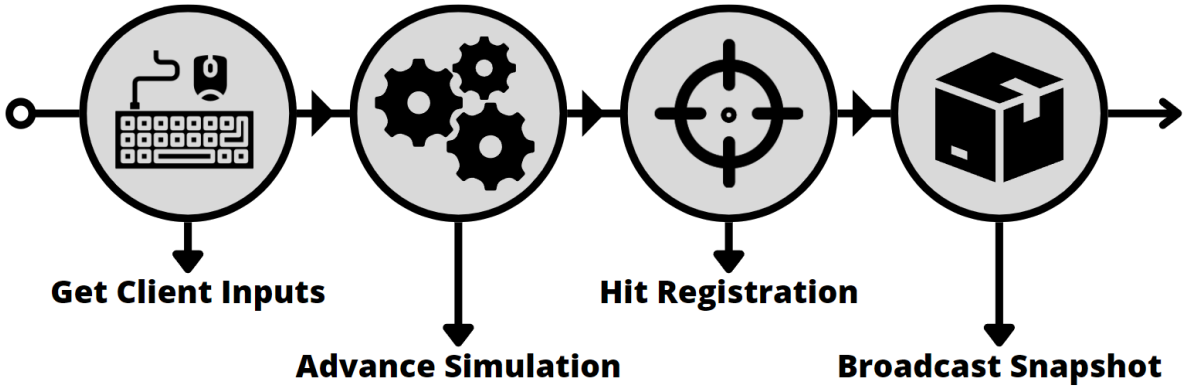


Ilustración 49 – Tareas llevadas a cabo durante un tick en el servidor

Obtención de inputs

El servidor realiza su primera tarea al obtener los *inputs* de los diferentes clientes en el juego. Este proceso se representa en la Ilustración 50. El objetivo principal es obtener ninguno, uno o varios *inputs* del Buffer de jitter de cada cliente conectado al servidor. Idealmente, este proceso devolverá un único *input* por cliente para simular en la siguiente tarea, aunque hay situaciones en las que esto no sucede. En ocasiones, el Buffer de jitter de un cliente puede estar vacío, lo que indica que no han llegado nuevos *inputs* para simular. En estos casos, el

servidor verifica si el número de *ticks* consecutivos en los que el *buffer* ha permanecido vacío supera un umbral determinado. En caso de no superarlo el servidor no realizará ninguna acción adicional y el estado de la entidad jugador de dicho cliente no será actualizado. Sin embargo, en caso de que se haya superado el umbral permitido, el servidor creará un *input* de reemplazo duplicando el último recibido por dicho cliente. En este punto, si el *input* original que debía de haberse usado llega al servidor, será automáticamente descartado puesto que ya no se considerará válido. El umbral máximo que se le permite al Buffer de jitter de un cliente permanecer vacío es actualmente de 2 *ticks* consecutivos. Este proceso de predicción de *inputs* puede perjudicar al cliente afectado, ya que puede dar lugar a una reconciliación la cual puede ser percibida por el jugador, especialmente si el servidor realiza múltiples predicciones consecutivas. Por otro lado, esta técnica beneficia al resto de jugadores, ya que aunque el servidor no reciba ningún *input* del cliente afectado, los demás clientes seguirán recibiendo actualizaciones del estado de dicha entidad, lo que les permite seguir realizando una interpolación fluida en sus máquinas.

Por otra parte, puede haber casos en los que este proceso devuelva varios *inputs* del Buffer de jitter para simular. Esto es debido a que cada Buffer de jitter posee una capacidad máxima recomendada, actualmente fijada en 2 *inputs*. Si se supera esta capacidad, el *buffer* comenzará a liberar *inputs* hasta que su tamaño deje de exceder este parámetro.

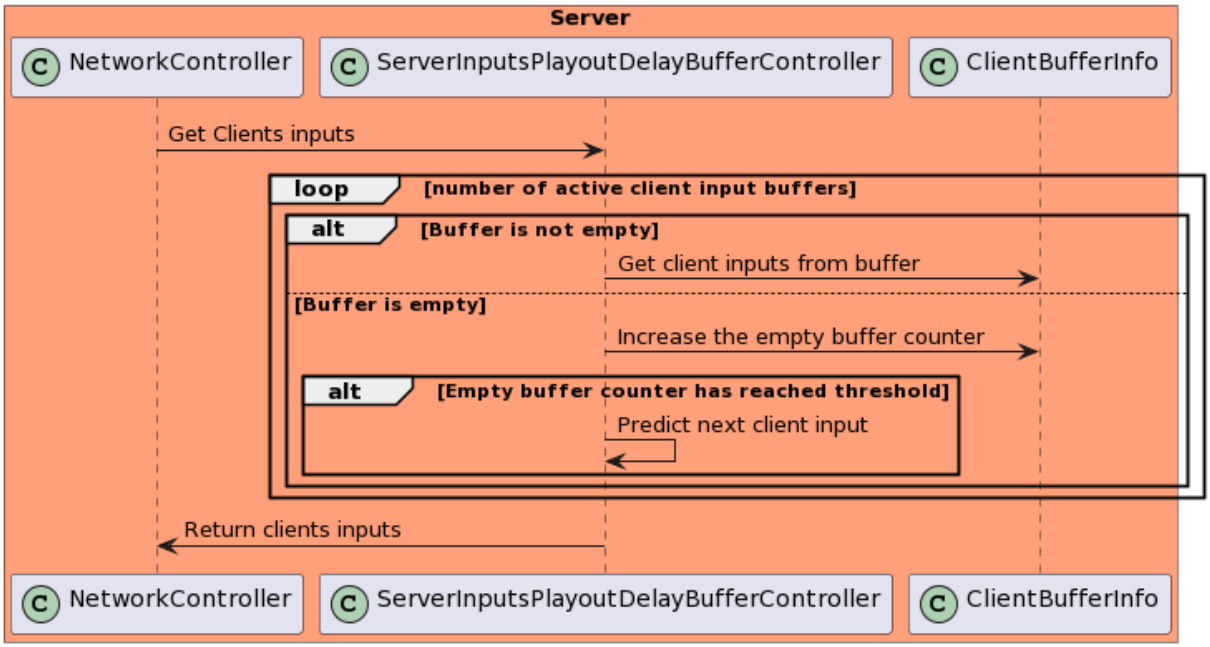


Ilustración 50 - Diagrama secuencia del proceso de obtención de inputs en el servidor para su posterior procesamiento

Avance de la simulación

Una vez obtenidos los *inputs* de todos los clientes estos serán transmitidos al componente encargado de realizar la simulación.

Este proceso de simulación se muestra ilustrado en la Ilustración 51 y consta de varios pasos:

1. En primer lugar, se simularán todas aquellas entidades que sean jugadores, es decir, que posean la interfaz de `INetworkPlayerEntity` y por lo tanto necesiten de un *input* para avanzar su estado.
2. Tras haber simulado y avanzado el estado de todas las entidades jugadores, se procederá a simular el resto de entidades. En este segundo paso, y como se ha explicado en un apartado anterior, el algoritmo de simulación no dependerá de ningún *input*.
3. Por último, se eliminarán todas aquellas entidades que hayan quedado inactivas durante el *tick* actual. Este proceso se realiza aparte puesto que eliminar elementos de una colección mientras se está iterando con un bucle puede ser problemático.

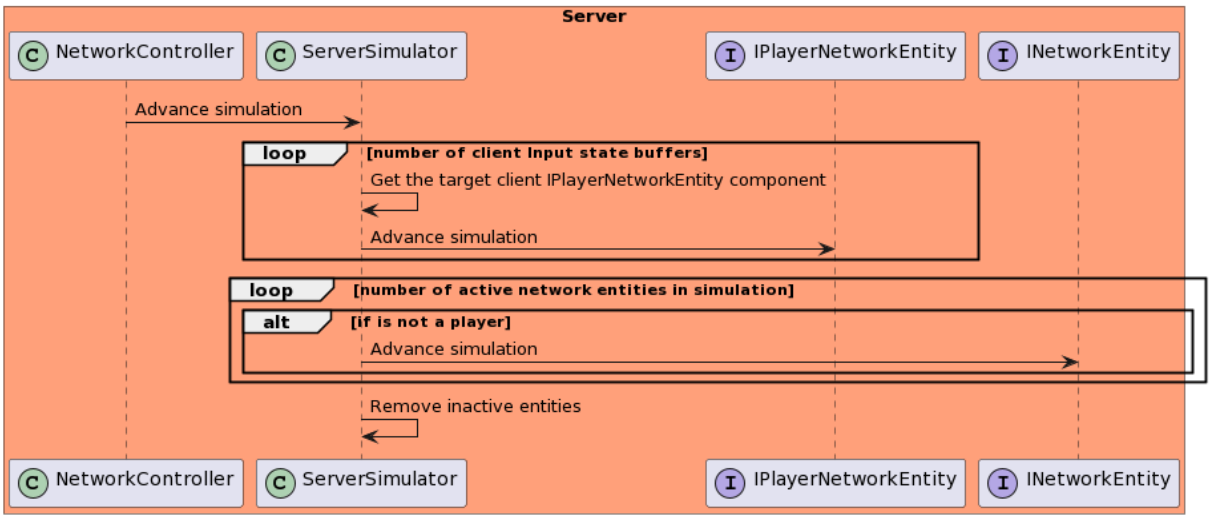


Ilustración 51 - Diagrama secuencia del proceso de simulación en el servidor

Hit registration

Durante el proceso de simulación, las distintas entidades podrán haber realizado la acción de disparar. Cuando esto ocurre, estas acciones se guardan en una cola conteniendo todas aquellas entidades que han disparado durante la simulación del *tick* actual. Tras este proceso, la tercera tarea del servidor trata de procesar todos estos disparos y determinar si han resultado exitosos o fallidos. Para ello, existe un componente específico encargado de realizar estos proceso el cual usa un algoritmo de *Hit Registration* junto con la técnica de compensación de la latencia explicada en el apartado de Registro de impactos. Este apartado valida el cumplimiento del requisito RF04

Las entradas de este algoritmo son el instante de tiempo actual del servidor y el conjunto de entidades activas en la simulación. Los pasos que sigue este algoritmo se ilustran en la Ilustración 52:

1. Mientras haya entidades dentro de la cola…
2. Coger el siguiente identificador de entidad de la cola.
3. Comprobar que dicho identificador pertenezca a una entidad válida de la simulación.

4. Calcular el instante de tiempo hasta el que hay que retroceder el estado de juego.
5. Retroceder el estado de todas las entidades que posean la capacidad de retroceder su estado de la simulación.
6. Obtener tanto el origen como la dirección del punto de disparo de la entidad atacante.
7. Calcular mediante operaciones de trazado de rayos el desenlace de dicho disparo.
8. En caso de que el disparo haya impactado con algún objeto intentar aplicarle daño y crear una entidad temporal para notificar a los clientes de este impacto.
9. Volver al paso 1 y comprobar si hay más entidades en la cola para procesar.
10. Deshacer el retroceso aplicado en el paso 4 para devolver a las entidades a su estado actual.

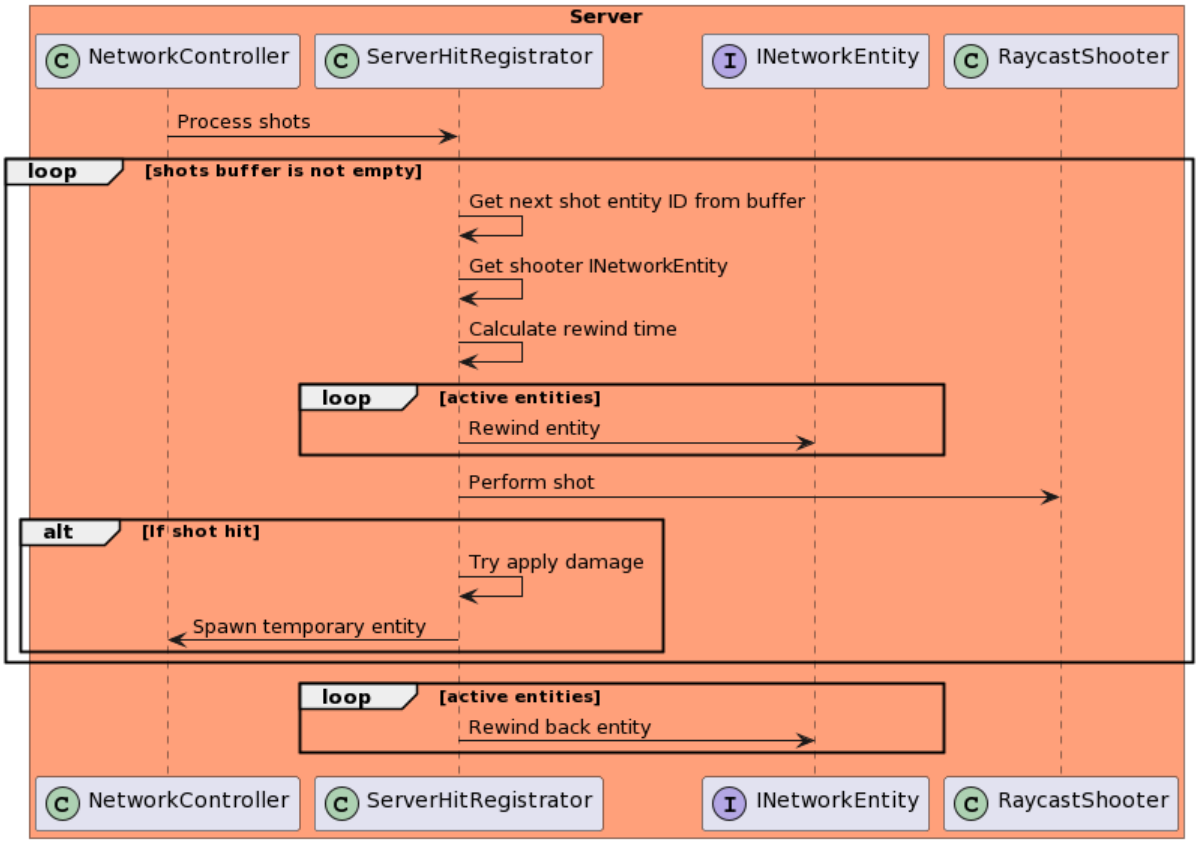


Ilustración 52 - Diagrama secuencia del proceso de Hit Registration en el servidor

Creación y envío de una snapshot

Por cada *tick* el proceso de creación de una *snapshot* se realiza tantas veces como clientes haya conectados en ese momento. Esto es debido a que cada *snapshot* enviada a un cliente es previamente personalizada por temas de optimización del tamaño de los paquetes. En primer lugar, cada *snapshot* solo puede almacenar un `Player state`. Esto es debido a que a cada cliente únicamente le es relevante obtener toda la información del jugador local que esté controlando, para poder continuar llevando a cabo los algoritmos de predicción y reconciliación en el lado del cliente. Para el resto de entidades, el cliente solo necesita un

`Entity State` para realizar las tareas de interpolación, el cual es una versión simplificado del estado total de una entidad (en este caso de tipo jugador).

Una vez personalizada esta será enviada mediante una llamada RPC únicamente al cliente destinatario.

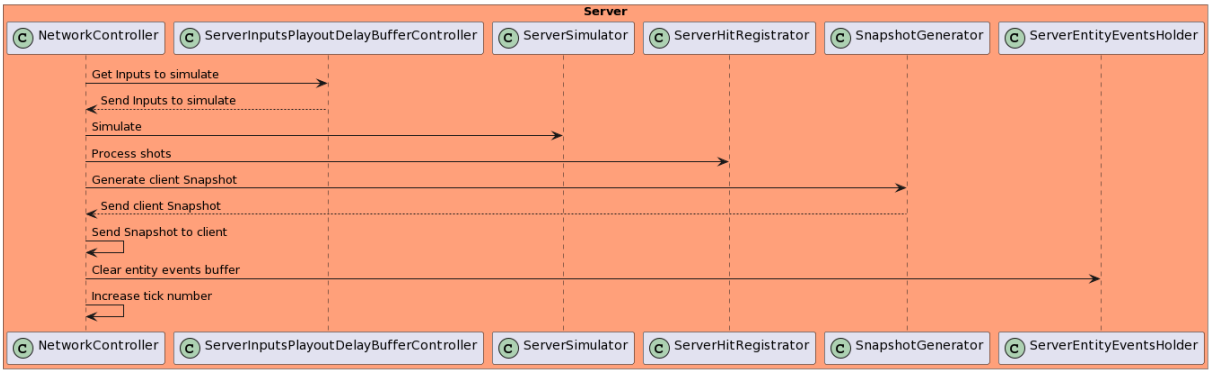


Ilustración 53 - Diagrama secuencia de las operaciones llevadas a cabo en el NetworkController durante un tick en el servidor

Llegada y procesamiento de *snapshots* en el cliente

El único sistema del `NetworkController` exclusivo de los clientes es el `ClientSnapshotProcessor.cs`. Este sistema es el encargado de procesar las *snapshots* recibidas por el servidor y distribuir su información entre las distintas entidades. Este proceso de distribución de la información, representado en la Ilustración 54, consta de dos pasos principales:

1. En primer lugar, se comprobará si la *snapshot* contiene un `Player state` válido. En caso afirmativo este será enviado a la entidad del jugador local mediante un método de la interfaz `IPlayerNetworkEntity`.
2. En Segundo lugar, se comprobará si existen `Entity states` válidas en el *array* de `Entity states`. En caso de no estarlo se buscará la entidad a la que corresponda cada `Entity state` y se le enviará mediante un método de la interfaz `INetworkEntity`.

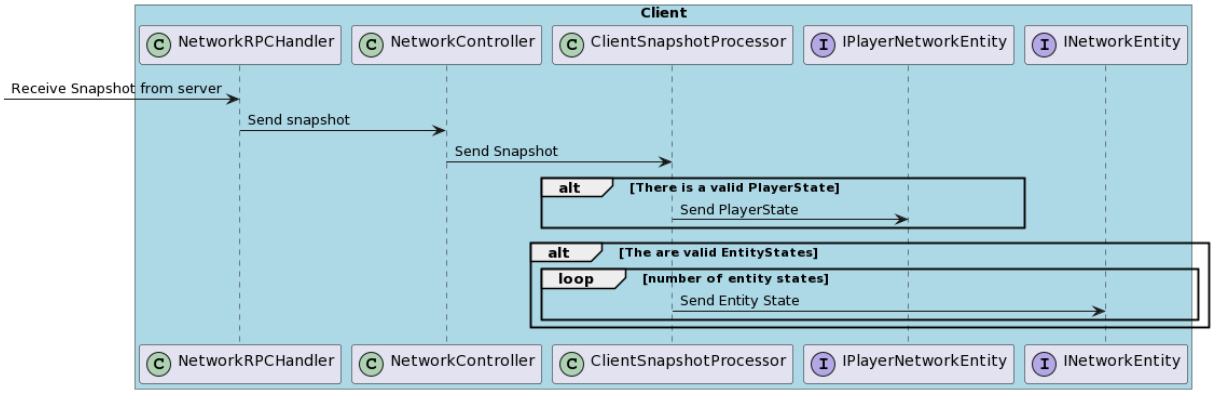


Ilustración 54 - Diagrama secuencia del proceso de procesamiento de una snapshot en el cliente

Entidad jugador

Proceso de simulación del jugador

La entidad jugador posee una serie de sistemas capaces de alterar el estado de este. En la Ilustración 55, se muestra un diagrama de flujo de las distintas operaciones llevadas a cabo en el proceso de simulación de la entidad jugador. Este proceso es el mismo tanto para un cliente como para un servidor.

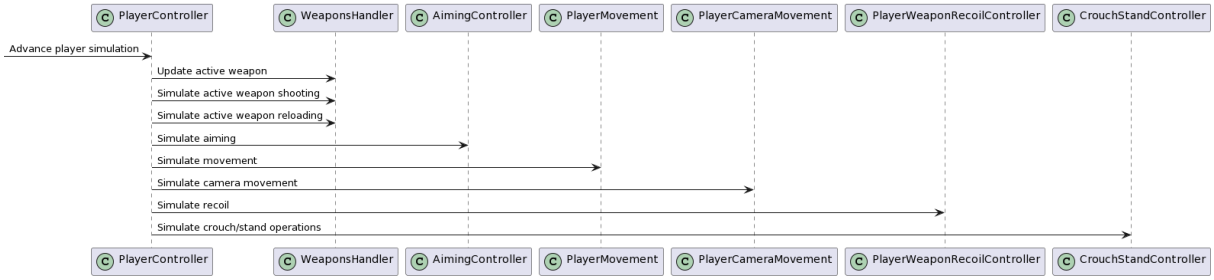


Ilustración 55 - Diagrama secuencia del proceso de simulación de una entidad jugador tanto para un cliente como para un servidor

Sistema de Ghosting en el jugador

El objeto jugador está dividido en dos partes fundamentales. En primer lugar, existe un objeto el cual se mueve y rota sin aplicarle ningún tipo de interpolación. Este objeto se le denomina G*host* y se mueve "a saltos". Además, posee *colliders* para llevar a cabo todo el proceso de detección de colisiones. Por otra parte los elementos visuales del jugador se encuentran en otro objeto el cual es el encargado de renderizar las animaciones o visualizar el mundo a través de la cámara del jugador. Este objeto persigue de forma continua y fluida al objeto *Ghost* del jugador mediante el uso de técnicas de interpolación. De esta forma, si se genera una reconciliación en el cliente, el objeto *Ghost* dará un salto hacia la nueva posición y rotación mientras que este objeto se irá alineando durante los próximos fotogramas. Este proceso de interpolación debe de no ser muy suave puesto que en ese caso se estaría generando una latencia adicional y podría afectar a otros sistemas como la precisión de los disparos, pero tampoco puede ser muy brusco ya que degradaría la experiencia de juego.

En la Ilustración 56 se muestra un ejemplo gráfico de cómo este sistema funciona. En él, existe un objeto *Ghost*, perteneciente al jugador local de un cliente, el cual va posicionándose cada *tick* en la nueva posición calculada tras el proceso de predicción en el lado del cliente. Además, existe un objeto interpolado el cual va siguiendo al objeto *Ghost* y cuya trayectoria puede verse representada mediante una curva continua azul. En el instante de tiempo 5, se recibe una corrección por parte del servidor de la posición generada en el instante número 3, provocando como consecuencia una reconciliación y una resimulación de los *ticks* 4 y 5. La posición resultante de este proceso se muestra con el número 5. Tras esto, el objeto *Ghost* se reposiciona mientras que el objeto interpolado deberá realizar esta corrección durante las siguientes actualizaciones en base a las futuras posiciones del objeto *Ghost*.

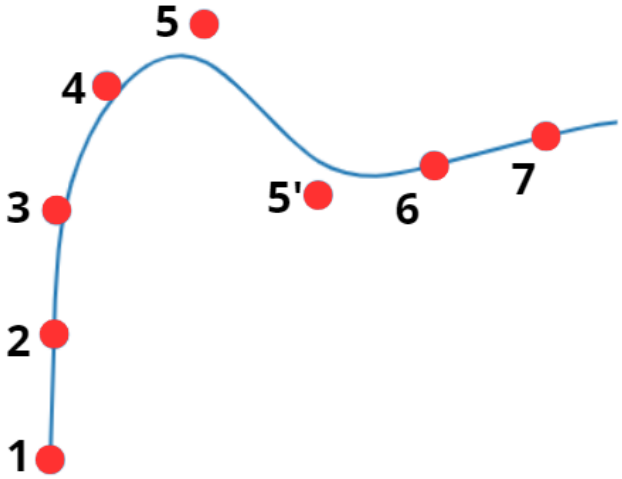


Ilustración 56 - Ejemplo gráfico del movimiento del objeto "Ghost" y el objeto interpolado de una entidad jugador

Este sistema de objeto *Ghost* y objeto interpolado únicamente está activo en el cliente al que posea como jugador local a dicha entidad jugador. En el servidor, la totalidad del jugador se moverá "a saltos" debido a que la finalidad de esta técnica es mejorar la fluidez del movimiento para los jugadores. Por otra parte, en caso de que la entidad jugador se ejecute en el cliente pero no sea su jugador local el sistema de *Ghosting* también permanecerá inactivo puesto que se usará un algoritmo de interpolación más complejo.

Proceso de predicción en el lado del cliente de estados futuros del jugador local

La predicción en el lado del cliente es llevada a cabo a través del sistema encontrado en el archivo `PlayerClientSidePredictor.cs`. En la Ilustración 57 se puede observar el proceso que sigue este algoritmo. En este proceso, se hace uso de *arrays* circulares para guardar un historial de *inputs* y estados de jugador en caso de que sea necesario volver a recuperarlos ante una posible reconciliación con el servidor. En este proceso de predicción en el lado del cliente se cumple el requisito RF03 así como el requisito RF01

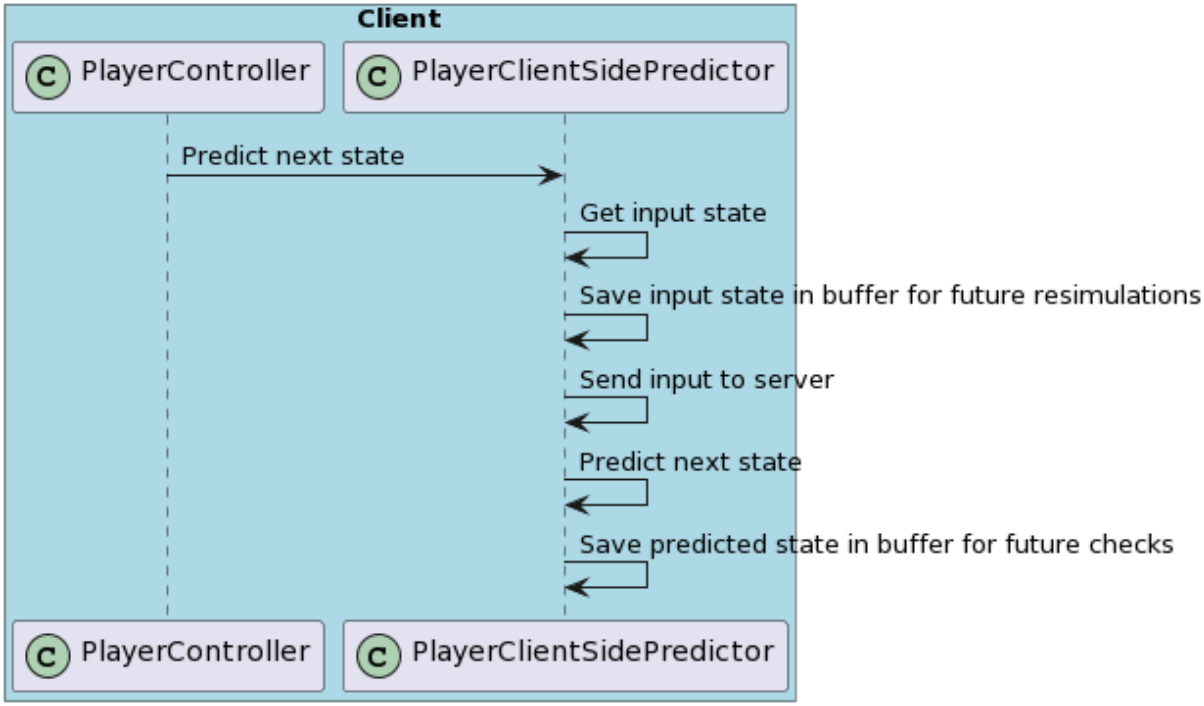


Ilustración 57 - Diagrama secuencia del proceso de predicción en el lado del cliente

El proceso de reconciliación con el estado del servidor y la posterior resimulación también se realiza en este sistema y cuyo proceso se representa de forma gráfica en la Ilustración 58. Este proceso consta de los siguientes pasos:

1. En cada fotograma se comprueba si se han recibido nuevos estados del jugador por parte del servidor.
2. En caso de que se cumpla el paso 1, se realiza una comparación entre dicho estado y el estado con el mismo identificador generado en el proceso de predicción en el cliente.
3. Si ambos estados difieren, el cliente sincronizará su estado con el estado recibido del servidor y comenzará el proceso de resimulación. Esta resimulación comenzará con el *input* inmediatamente posterior al estado recibido por el servidor hasta haber resimulado el último *input*.

Este proceso de reconciliación con el estado del jugador en el servidor cumple con el requisito RF08.

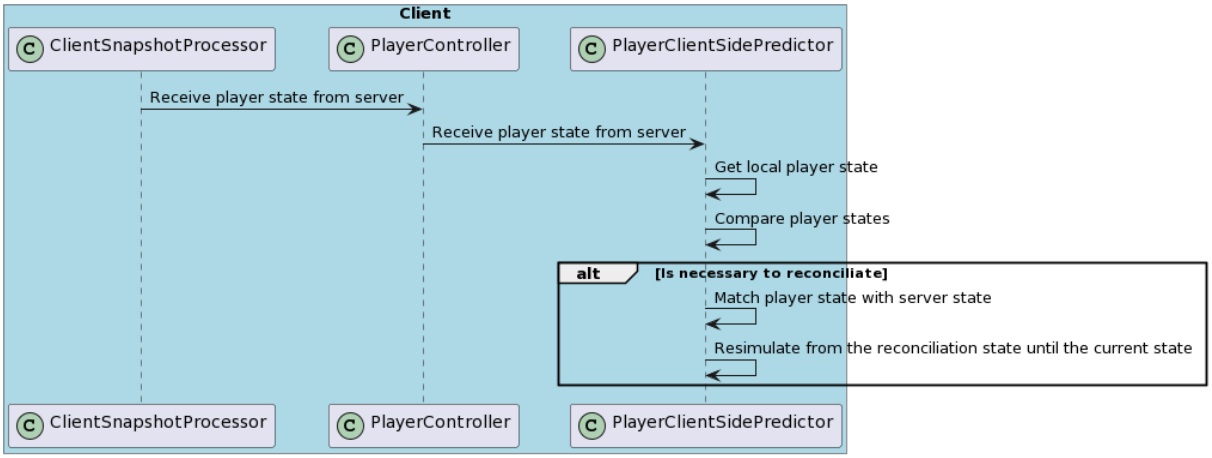


Ilustración 58 - Diagrama secuencia del proceso de reconciliación con el estado del servidor y su posterior resimulación

Inicialización personalizada del jugador

La entidad jugador es la única entidad que implementa la interfaz `IPlayerNetworkEntity`. Además, al igual que el `NetworkController`, la entidad jugador posee la capacidad de personalizar la inicialización de sus componentes dependiendo de varios factores. Hay tres posibles casos de inicialización distintos. (i) Si la entidad jugador pertenece al jugador local del cliente, (i) si es una entidad jugador en el servidor o (iii) si es una entidad jugador que no pertenece al jugador local del cliente. A continuación se detallará de qué forma se inicializan los distintos sistemas del jugador dependiendo de estos casos.

En primer lugar, los sistemas necesarios para el avance de la simulación serán inicializados únicamente si la entidad jugador se crea en el servidor o si se crea en el cliente como jugador local. Estos sistemas son los mostrados en la Ilustración 55.

Por otra parte, el sistema encargado de llevar a cabo la predicción y reconciliación en el lado del cliente únicamente será inicializado si la entidad jugador se crea en el cliente como jugador local.

Los sistemas encargados de manejar los modelos de primera y tercera persona también se inicializarán de manera diferente. En el servidor, el sistema que maneja el modelo de primera persona, que incluye los brazos del jugador, estará completamente inactivo, mientras que el sistema del modelo de tercera persona solo activará las animaciones y los *colliders*. En el cliente, el modelo de primera persona solo estará activo si el jugador es local, y el modelo de tercera persona solo estará activo si no es el jugador local.

Interpolación y extrapolación de entidades

Para la implementación de la interpolación de entidades en el cliente se ha usado la técnica de interpolación explicada en el apartado Interpolación de entidades en el cliente.

Cuando la entidad, la cual implementa la interfaz `INetworkEntity`, recibe un nuevo estado de tipo `EntityState` por parte del servidor mediante el método `ReceiveEntityState`, esta comprobará si dicho estado es válido y en caso afirmativo lo introducirá en el Buffer de

jitter dedicado a la interpolación para su posterior uso. Un estado será válido únicamente cuando el instante de tiempo en el que se generó sea posterior al instante de tiempo del estado objetivo al que dicha entidad se encuentre interpolando en ese momento. Al añadirlo al *buffer* se calculará la posición dentro del mismo en la que debería ir situado. En la mayor parte de los casos esta posición será al final del resto de estados ya almacenados, pero si algún estado generado posteriormente ha llegado antes al cliente, este deberá de ser reposicionado. Gracias a este ajuste se disminuirán el número de estados inválidos ya que a pesar de que no ser el estado más posterior, si la interpolación todavía no ha llegado a ese instante es posible almacenarlo.

Por otra parte el algoritmo de interpolación posee un *offset* de tiempo calculado con la siguiente fórmula: $INTERPOLATION\ DELAY\ TICKS * (\frac{1}{TICK\ RATE})$ donde $INTERPOLATION\ DELAY\ TICKS$ representa el número de *ticks* de retraso que el cliente necesita esperar para empezar a interpolar. Para el caso de este prototipo el retraso es igual a 3 *ticks*. Si este número se multiplica por el tiempo transcurrido entre dos *ticks* consecutivos $1/50$ segundos, se obtendría un retraso igual a 60 milisegundos. Tras realizar una serie de pruebas se ha deducido que un retraso de 3 *ticks* no genera apenas artefactos visuales en latencias aceptables ni un gran retraso que pueda llegar a afectar a la precisión de los disparos del jugador. La implementación de estas técnicas de interpolación en el prototipo cumple con el requisito RF05.

La implementación de la extrapolación de entidades en el cliente se ha llevado a cabo usando el algoritmo de *Projective Velocity Blending* (Mezcla de velocidades proyectadas) explicado en el apartado de Extrapolación de entidades en el cliente. Con esta implementación se consigue cumplir el requisito RF06.

Capítulo 5

Validación

En este apartado de validación, se presenta un análisis exhaustivo de los distintos *tests* realizados durante este proyecto. Estos *tests* abarcan diversos aspectos del prototipo como pruebas unitarias, procesos de *Beta testing* y la evaluación de algunos sistemas de crucial importancia. Los resultados obtenidos de todos estos *tests* validan la funcionalidad y experiencia del prototipo, confirmando la efectividad de las soluciones implementadas.

6.1 Pruebas unitarias

A lo largo del desarrollo de este prototipo multijugador, se han ido creando diferentes *tests* con la finalidad de evaluar distintas funcionalidades y componentes de este proyecto. La mayoría de los *tests* realizados se han centrado en la validación de los distintos componentes clave del prototipo, muchos de los cuales estaban relacionados con el apartado de *netcode*. Por ejemplo, entre estos *tests* se pueden destacar aquellos encargados de la validación del componente de conexión tanto en el lado del cliente como en el del servidor. La validación de este componente se ha llevado a cabo mediante la ejecución de varios *tests* de tipo *PlayMode* en Unity Engine. Algunos ejemplos de estas pruebas han sido las siguientes:

1. Enviar una petición de conexión desde el cliente al servidor comprobando que esta petición resulta exitosa y, como consecuencia, el cliente acaba conectándose correctamente.

2. Repetir el test anterior y desconectar al cliente que acaba de ser conectado para comprobar que este proceso de desconexión se lleva a cabo de forma exitosa.

3. Repetir el test anterior, volviendo a conectar de nuevo al cliente que se ha desconectado para comprobar si esta segunda conexión ha sido correcta. De esta forma, el sistema es capaz de comprobar si tras la primera desconexión todo se ha reiniciado correctamente para que la próxima vez que se conecte el cliente vuelva a empezar de cero.

6.2 Evaluación de los algoritmos de interpolación y extrapolación

Descripción del test

Aparte de los *tests* mencionados anteriormente, se ha llevado a cabo otro test el cual tiene como objetivo estudiar y validar el correcto funcionamiento de los algoritmos de interpolación y extrapolación de entidades, implementados en el lado del cliente así como demostrar las distintas mejoras que estas técnicas proporcionan.

Como ya se ha explicado anteriormente en los apartados de Interpolación de entidades en el cliente y Extrapolación de entidades en el cliente, estas técnicas buscan mitigar los problemas de la red para conseguir proporcionar un movimiento fluido para estas entidades, sobre todo en los casos en los que el *tickrate* del servidor sea menor a la tasa de fotogramas por segundo a la que el videojuego renderiza el estado del juego.

Para este *test,* se ha grabado una secuencia de *inputs* reales en un fichero de texto para poder reproducirlos una y otra vez en los distintos casos que conforman este *test*. De esta forma, se podrán comparar los resultados generados a partir de los mismos datos de entrada. Además, para cada caso de estudio, tanto el servidor como el cliente han guardado en archivos de texto diferentes el historial con las distintas posiciones que una entidad jugador ha ido recorriendo en cada una de las máquinas así como el instante de tiempo en el que esas posiciones han sido aplicadas al estado del juego. En el servidor, estas posiciones se han ido recogiendo a una tasa de 50 veces por segundo, que es la frecuencia con la que, el servidor elegido para este *test,* actualiza el estado del juego (*tickrate*). Por otra parte, el cliente ha ido almacenando las posiciones generadas por los sistemas de interpolación y extrapolación de esa entidad jugador. Esta frecuencia de escritura ha sido algo mayor que la del servidor puesto que estos sistemas se ejecutan en la función *Update* proporcionada por Unity Engine, la cual no tiene una frecuencia fija.

El prototipo ha sido evaluado con distintos valores de RTT: 40, 80, 160 y 320 milisegundos. Para cada uno de esos valores de RTT, se ha probado el prototipo con niveles de *jitter* y pérdida de paquetes del 0%, 2%, 4% y 8%. Además, para cada uno de estos niveles se han hecho dos pruebas diferentes: una con los sistemas de interpolación y extrapolación de entidades en el cliente activados y otra con ellos desactivados.

Tanto el RTT, el *jitter* o la pérdida de paquetes en los diferentes casos de prueba de este *test* han sido simulados de forma artificial mediante el uso de la herramienta Clumsy.

Una vez obtenidos ambos ficheros con las distintas posiciones de la entidad jugador tanto por parte del cliente como del servidor, se ha pasado a graficar para cada caso de estudio las diferencias entre las posiciones del cliente y del servidor lo largo de todo el recorrido.

Resultados obtenidos

Los resultados se encuentran graficados en el Anexo I Gráficas de la evaluación de los algoritmos de interpolación y extrapolación

Discusión de los resultados

En este subapartado se van a discutir los resultados obtenidos en este *test*.

Desventaja de los algoritmos de interpolación

Al comparar las discrepancias entre las posiciones del cliente y del servidor a lo largo del recorrido, se observa una primera diferencia. Las gráficas muestran que, cuando los sistemas de interpolación están activos, la discrepancia entre las posiciones, la cual está representada por el eje vertical, son mayores que cuando estos sistemas no están activos. Por ejemplo, al observar las gráficas representadas en la Ilustración 59, Ilustración 60 y la Ilustración 61, se puede apreciar que tanto la media global (representada con una recta horizontal discontinua), como la media móvil (representada por una curva), poseen valores menores en el eje vertical, es decir, en la diferencia de posiciones entre el cliente y el servidor, para el caso en el que no se utilizan los sistemas de interpolación de entidades en el cliente (representado en color rojo), en comparación con el caso en el que sí se utilizan (representado en color azul).

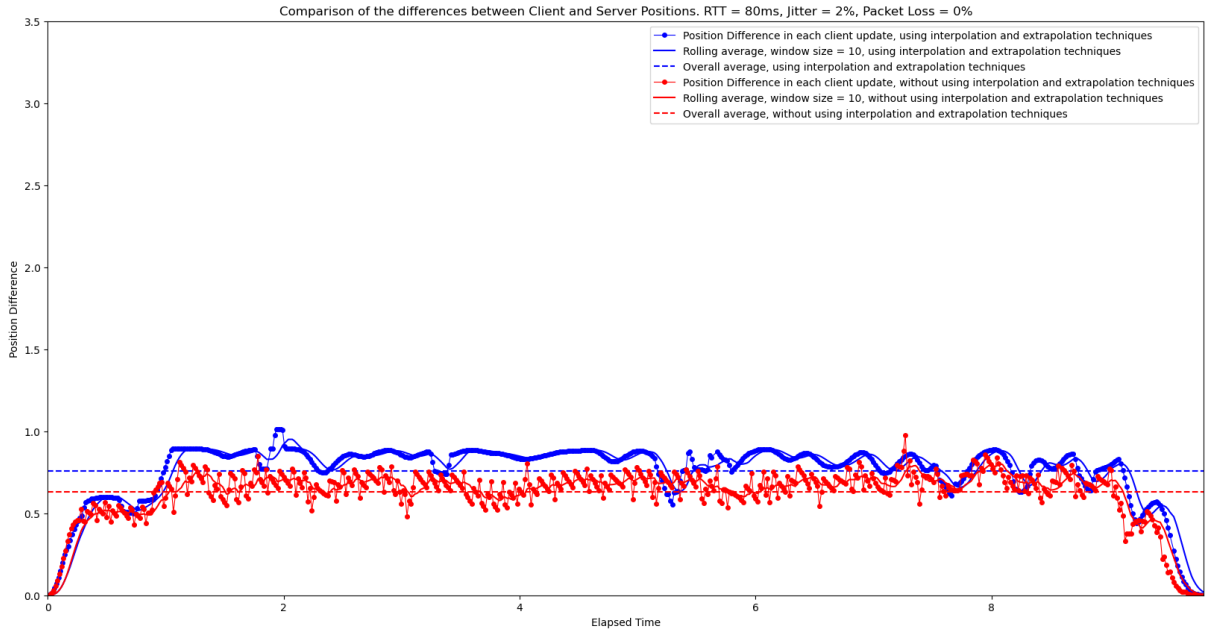


Ilustración 59 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 80ms usando un porcentaje de 2% de jitter y 0% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.

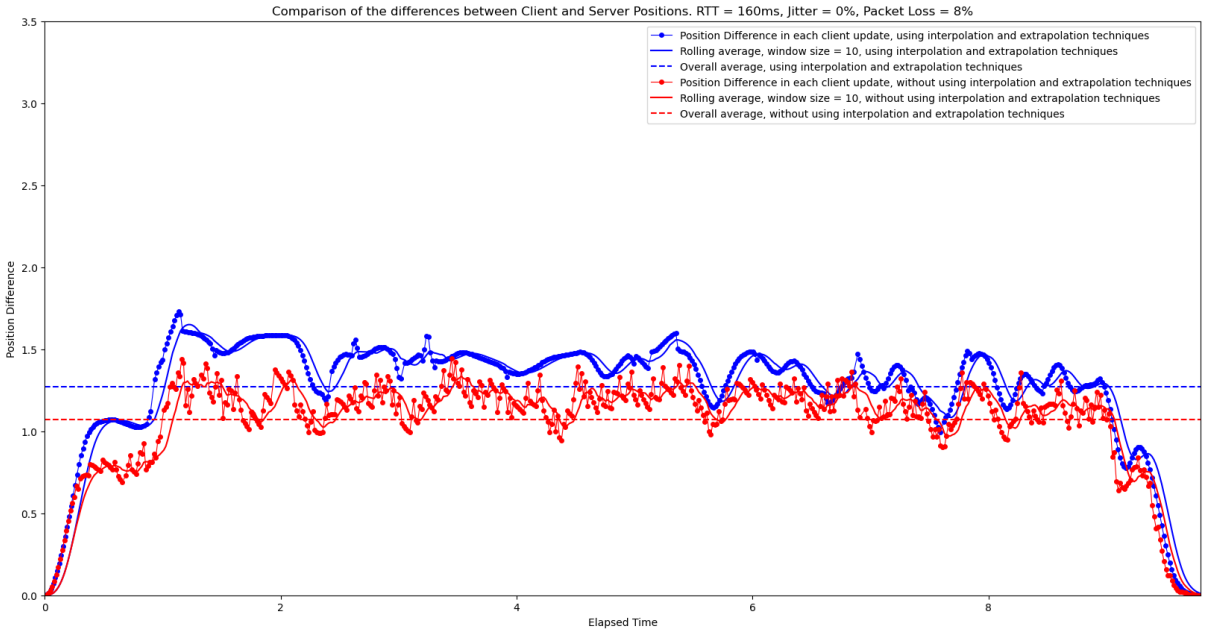


Ilustración 60 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 160ms usando un porcentaje de 0% de jitter y 8% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.

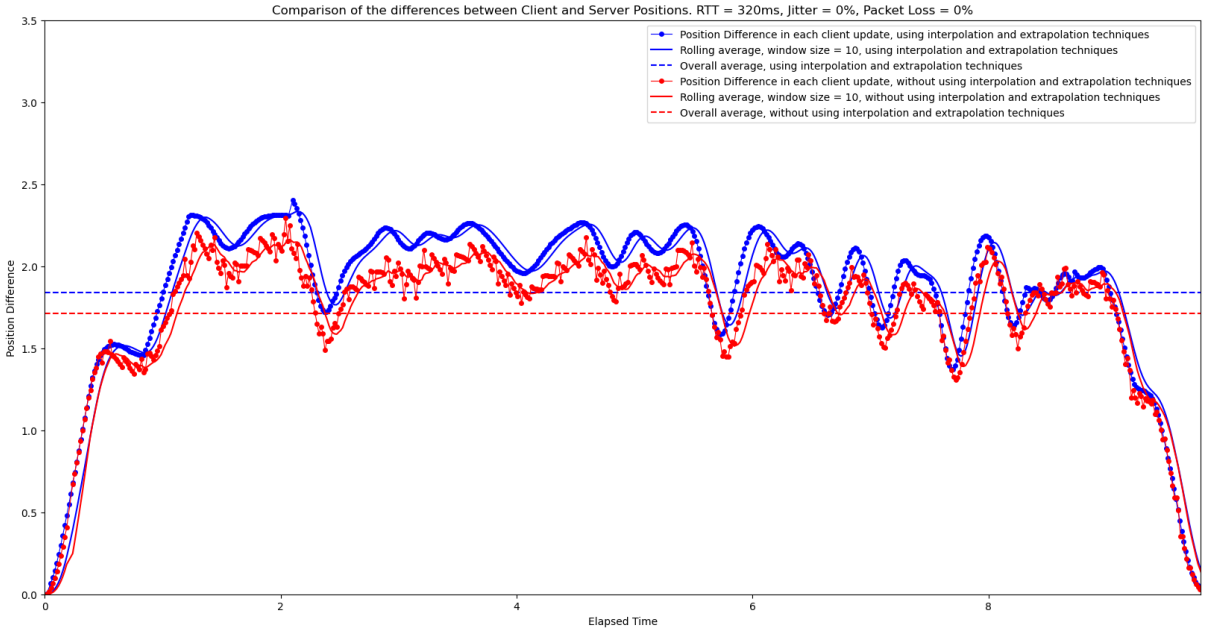


Ilustración 61 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 320ms usando un porcentaje de 0% de jitter y 0% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.

Además, independientemente de los valores de RTT, *jitter* y/o pérdida de paquetes, la diferencia entre ambas medias globales siempre es aproximadamente de 0.2 unidades. Este aspecto es un comportamiento esperado debido al funcionamiento de estos algoritmos de

interpolación. La diferencia de posiciones entre los casos donde se usan los sistemas de interpolación y los casos donde no se usan es causada por el *buffer* de *jitter* que estos algoritmos utilizan. Como ya se comentó anteriormente en el apartado de Interpolación y extrapolación de entidades, la latencia que el *buffer* de *jitter,* usado en este prototipo, añade es de 3 *ticks* del servidor lo que supone un retardo adicional de 60 milisegundos en el lado del cliente. Como consecuencia, mientras los estados de las entidades remotas provenientes del servidor se almacenan en el *buffer* de *jitter* a la espera de ser aplicados en el cliente, el servidor continúa avanzando la simulación alejándose cada vez más del estado renderizado por el cliente en ese momento.

Ventaja de los algoritmos de interpolación

Este retardo adicional a la hora de renderizar los distintos estados de las entidades remotas en el cliente es el único problema que se observa en los casos donde estos sistemas de interpolación permanecen activos. Sin embargo, se puede apreciar una ventaja significativa en la fluidez de las diferencias entre las posiciones del cliente y el servidor.

Al comparar gráficas como aquellas representadas en la Ilustración 62 y en la Ilustración 63, donde se muestran las diferencias de posiciones entre el cliente y el servidor con y sin los sistemas de interpolación activos, concretamente para los casos en los que el RTT es de 320 milisegundos y los porcentajes de pérdida de paquetes son de 0%, 2%, 4% y 8% (en base al color), se puede apreciar un cambio significativo en la fluidez de estas diferencias de posiciones. Mientras que en la Ilustración 62, la cual usa interpolación, las medias móviles para todos los porcentajes de pérdida de paquetes son curvas continuas, sin ningún cambio de dirección abrupto, en la Ilustración 63, la cual no usa la interpolación, las medias móviles están llenas de picos y cambios de dirección abruptos.

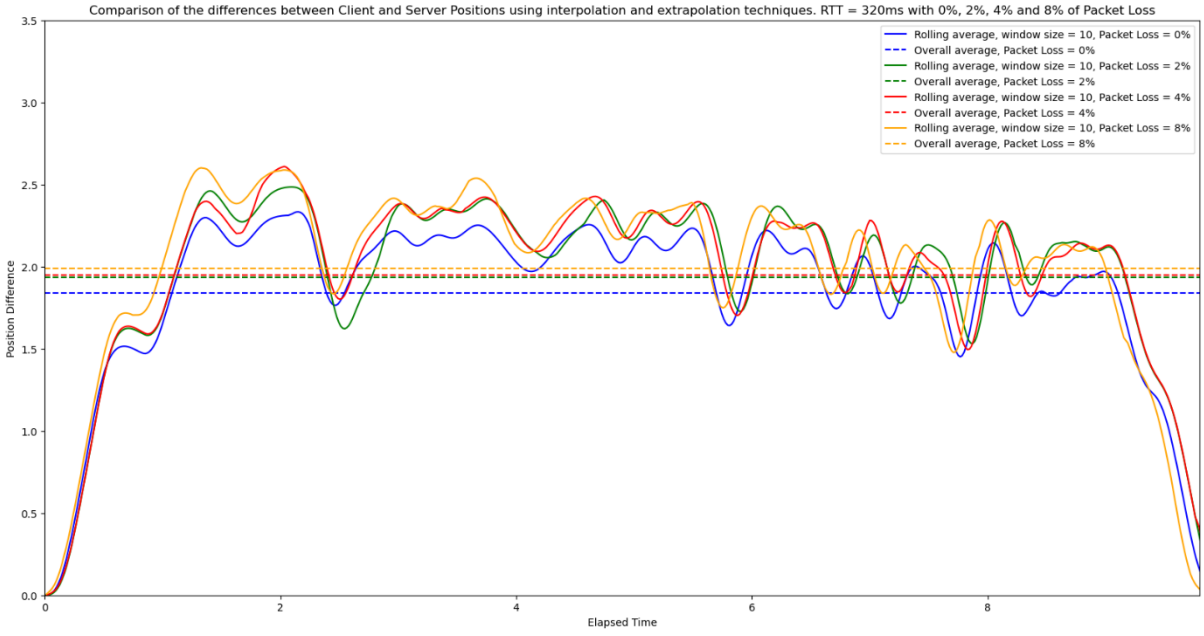


Ilustración 62 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de

entidades en el cliente con un RTT de 320 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

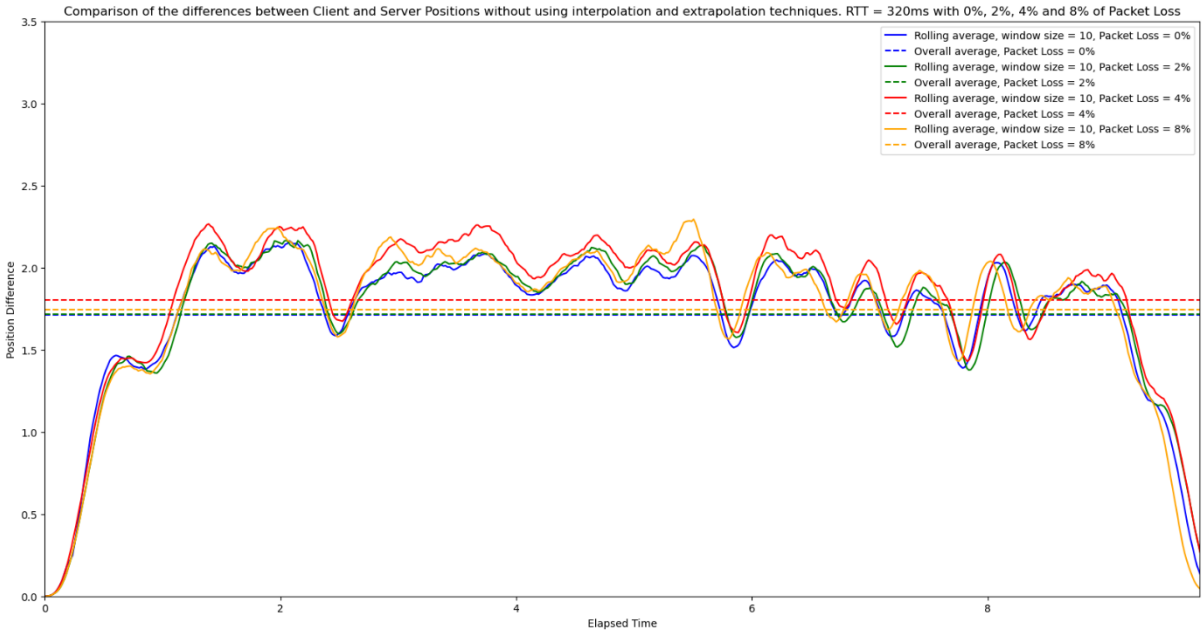


Ilustración 63 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 320 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

Por otra parte, en la gráfica representada en la Ilustración 92Ilustración 64 se puede observar que para el caso donde no hay interpolación (representado en rojo), los diferentes puntos consecutivos de la gráfica están muy dispersos entre sí y muy rara vez aparecen juntos formando un camino continuo, a diferencia del caso con interpolación (representados en azul).

La causa por la que esta dispersión de puntos y picos en las medias móviles ocurre radica en la propia interpolación. En los casos en los que no se usan los sistemas de interpolación, cuando una nueva posición llega al cliente esta se aplica de manera instantánea, reduciendo así la diferencia de posiciones entre el cliente y el servidor drásticamente. Esto explica por qué las gráficas coloreadas en rojo como la representada en la Ilustración 64 muestran cambios bruscos entre los distintos puntos consecutivos del recorrido y como resultado picos en la curva que representa la media móvil, distorsionando la representación.

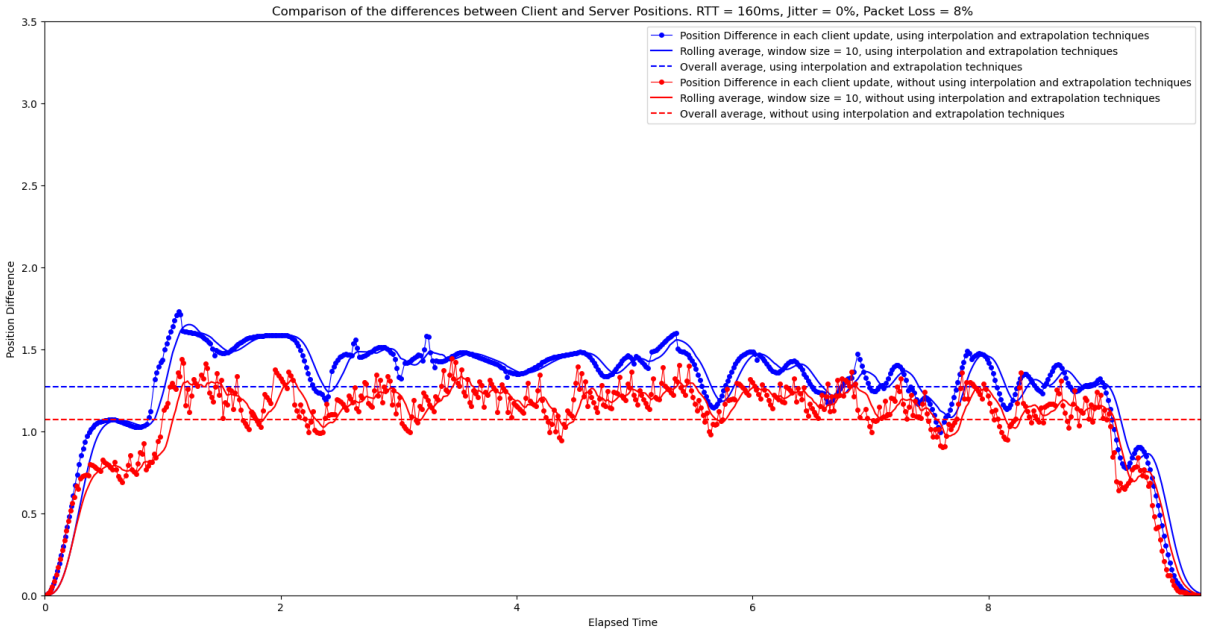


Ilustración 64 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 160ms usando un porcentaje de 0% de jitter y 8% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.

Por otro lado, en aquellos casos en los que se emplean los algoritmos de interpolación, la posición no se aplica de manera instantánea, sino que el cliente se aproxima a la posición del servidor de forma progresiva mediante la generación de estados y posiciones intermedias, de ahí el nombre de algoritmo de interpolación. Esta es otra razón por la cual es posible apreciar una mayor cantidad de puntos en aquellos casos donde se aplican los algoritmos de interpolación en comparación a aquellas gráficas donde no se aplican.

Debido a estas dos razones explicadas y corroboradas gráficamente, se puede afirmar que los algoritmos de interpolación de entidades en el lado del cliente proporcionan un movimiento mucho más suave y fluido a lo largo del tiempo mitigando así la aparición de ciertos artefactos visuales como pequeños saltos instantáneos.

El uso de un algoritmo de extrapolación

Son pocos los casos en los que, gráficamente, la extrapolación de entidades en el lado del cliente se manifiesta en un salto brusco en las diferencias de posiciones entre el cliente y servidor. Estos saltos ocurren cuando la posición extrapolada del cliente se aleja de la simulada por parte del servidor.

Estas desviaciones suelen ser mucho más probables en aquellos tramos del recorrido en los que existe una curva, puesto que los algoritmos de extrapolación implementados en este prototipo solo consideran la posición y la velocidad lineal en el cálculo de la nueva posición. Si se quisiera lograr una extrapolación más precisa y robusta ante este tipo de tramos, el algoritmo de extrapolación necesitaría considerar, aparte de lo ya mencionado, la velocidad y aceleración angular. Sin embargo, agregar datos adicionales a este cálculo conlleva un aumento del tamaño del paquete. Incluir la velocidad y aceleración angular implicaría

incorporar ambos vectores dentro de la *snapshot* para cada una de las entidades. Aunque el uso de algoritmos de compresión podría reducir este impacto, no es el enfoque actual de este prototipo.

Por otra parte, cuanto peores sean los valores de *jitter* y pérdida de paquetes, mayor será la probabilidad de que este tipo de situaciones ocurran. Un ejemplo de esta afirmación puede verse si se comparan la gráfica de la Ilustración 65 y la Ilustración 66Ilustración 89. Ambas gráficas poseen los algoritmos de interpolación y extrapolación activados. Sin embargo, se observa que aquella en la que se ha visto afectada por un 8% (Ilustración 66) de *jitter* posee un mayor número de saltos bruscos para dos puntos consecutivos que en aquella en la que se está aplicando un 0% de *jitter* (Ilustración 65). De hecho, a medida que las condiciones de red empeoran, estas gráficas comienzan a asemejarse más a los casos en los que no hay interpolación (p.ej. Ilustración 67 Ilustración 86o Ilustración 68) puesto que se producen más reconciliaciones, causadas por extrapolaciones erróneas.

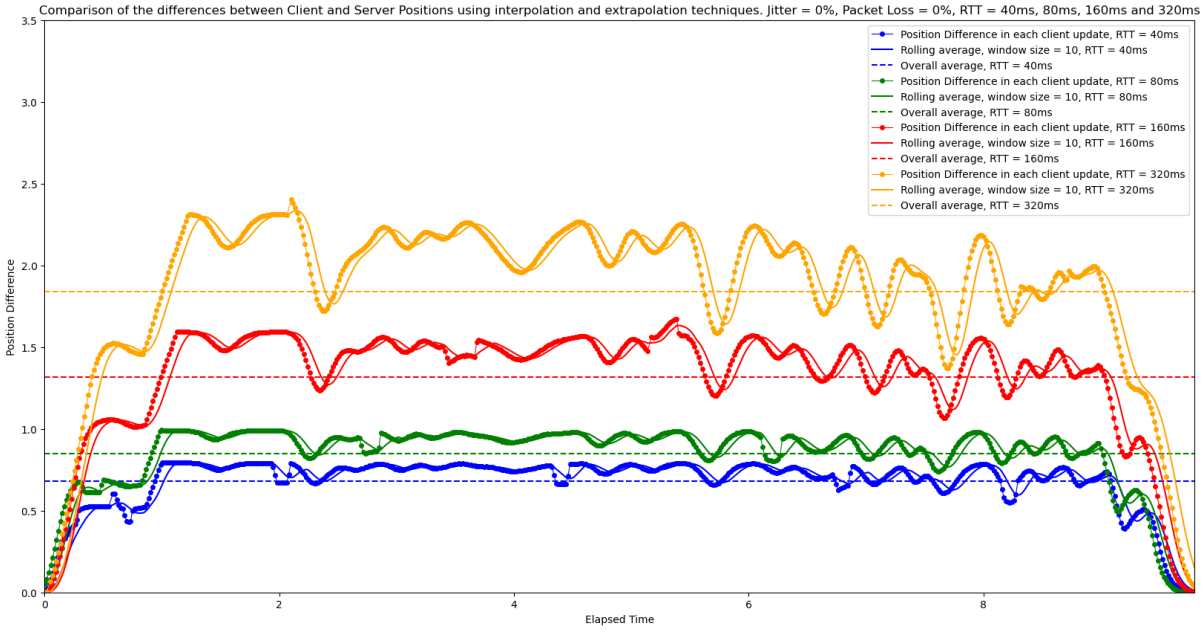


Ilustración 65 - Comparación de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 0% de jitter y 0% de pérdida de paquetes.

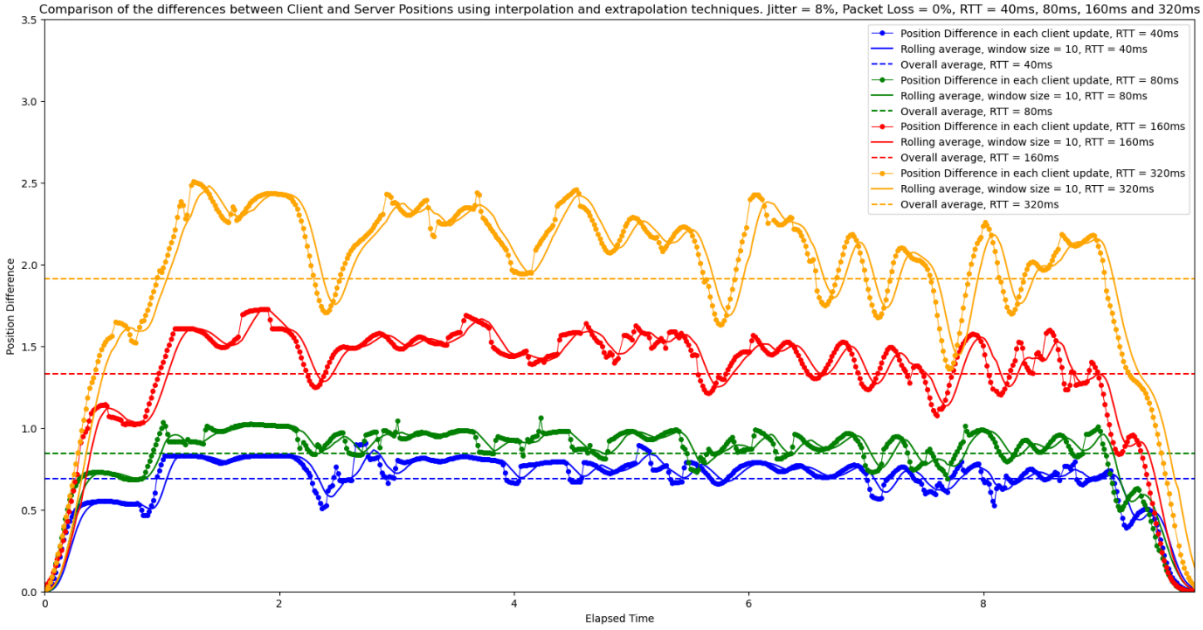


Ilustración 66 - Comparación de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 8% de jitter y 0% de pérdida de paquetes.

Por otra parte, debido al uso del algoritmo de *Projective Velocity Blending* explicado en el apartado de Extrapolación de entidades en el cliente, el proceso de reconciliación se realiza de forma fluida y progresiva siempre y cuando la diferencia de posiciones generada no exceda un cierto umbral. De esta forma, a pesar de que el algoritmo de extrapolación haya predicho de forma errónea alguna posición futura, se consigue evitar, en muchos casos, esos pequeños saltos instantáneos que se producen constantemente en aquellos casos en los que los sistemas de interpolación y extrapolación se encuentran desactivados (p.ej. Ilustración 67Ilustración 86 o Ilustración 68).

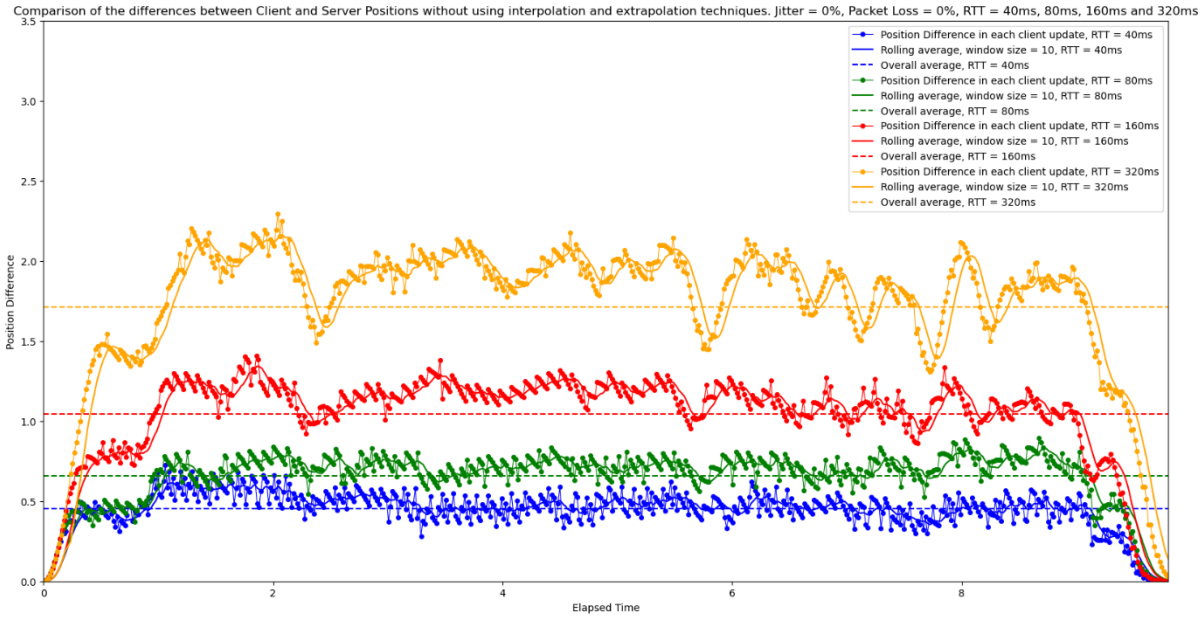


Ilustración 67 - Comparación de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 0% de jitter y 0% de pérdida de paquetes.

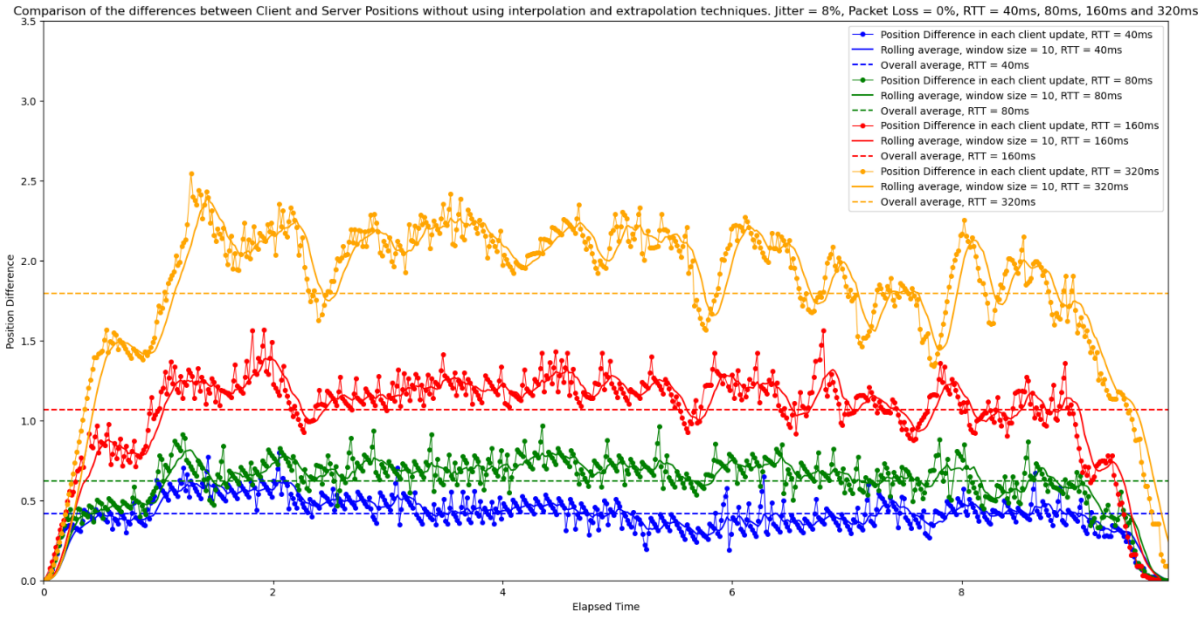


Ilustración 68 - Comparación de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 8% de jitter y 0% de pérdida de paquetes.

El uso de otros sistemas que repercuten de forma indirecta en la calidad de los resultados de la interpolación

Como ya se ha comentado anteriormente, cuanto menor sea la frecuencia de uso de técnicas de extrapolación, mayor será la calidad de la experiencia final. Para reducir esta frecuencia de uso, el prototipo posee una técnica implementada en el lado del servidor que mitiga parte de los problemas causados por el *jitter* y la pérdida de paquetes de la red.

Como se explicó en el apartado de Llegada de un *input* al servidor, en este prototipo se ha implementado una técnica que mitiga los efectos del *jitter* y la pérdida de paquetes, al predecir aquellos *inputs* que no han llegado a tiempo o no han alcanzado el servidor. En caso de que a la hora de simular el estado del juego no exista ningún *input* almacenado para un cliente en particular, el servidor predecirá un nuevo *input* duplicando el último que tenga registrado por ese cliente, con la finalidad de usarlo en el proceso de simulación del *tick* actual. Esta decisión puede modificar la trayectoria final del jugador si el *input* predicho difiere del *input* que tendría que haber llegado. Es por ello que, cuanto peores sean las condiciones de red, mayor será el número de predicciones que el servidor debe de realizar para mitigar los problemas del *jitter* y la pérdida de paquetes de la red, y como consecuencia, mayor será la probabilidad de que esta predicción difiera del *input* transmitido por parte del cliente.

Este sistema de predicción de *inputs* en el lado del servidor tiene grandes beneficios para el sistema de interpolación y extrapolación de entidades implementado en el cliente. Esto se

debe a que el sistema de predicción de inputs mitiga los problemas que puedan ser causados durante la comunicación desde el cliente hasta el servidor dejando únicamente en manos de los sistemas de interpolación y extrapolación aquellos problemas causados durante la comunicación desde el servidor hasta el cliente.

6.3 Beta testing

En el proceso de *beta testing* se ha reunido a un grupo de jugadores para probar el prototipo desarrollado en su estado actual a cambio de que puedan realizar comentarios y aportar una retroalimentación sobre su rendimiento, funcionalidad y eficacia de las técnicas implementadas en el mismo. El objetivo de este *beta testing* es identificar posibles vulnerabilidades y errores en la experiencia del jugador, la estabilidad o el rendimiento en distintas máquinas así como la seguridad del proyecto para mejorarlo en fases posteriores.

Para este *beta testing*, se ha elegido a una serie de conocidos del desarrollador, intentando buscar distintos niveles de habilidad en el género de los videojuegos de disparos en primera persona y con diferentes configuraciones tanto de conexiones a internet como de hardware.

La ejecución de este *beta testing* consistió en varias sesiones de juego de pocos minutos de duración en las que distintos jugadores se conectaban e intentaban acabar con los distintos enemigos.

Tras estas sesiones de juego se pudieron sacar distintas conclusiones:

1. Percepción de la latencia: En esta fase de *beta testing* se crearon dos ejecutables del mismo juego. En el primero de ellos, la interfaz de usuario exponía una serie de indicadores visuales de telemetría entre los que se encontraba un indicador del RTT de cada cliente con el servidor. Por otra parte, el segundo ejecutable ocultaba estos indicadores. Tras realizar varias sesiones de juego de corta duración con estos dos ejecutables y, tras los comentarios recibidos por parte de los participantes, se pudo llegar a la conclusión que ante latencias de carácter medio (en el rango de 45ms – 60ms) aquellos jugadores con el primer ejecutable, el cual poseía los indicadores de telemetría visibles, eran más conscientes de los problemas causados por esa latencia que aquellos que jugaban en el segundo ejecutable. Estos problemas, ordenados de mayor a menor frecuencia, eran: (i) errores de predicción en el lado del cliente debido a la predicción de *inputs* en el servidor o (ii) errores producidos por el algoritmo de extrapolación de entidades en el lado del cliente. Como consecuencia, la experiencia de juego de los jugadores del primer ejecutable se veía ligeramente más perjudicada que la de aquellos con el segundo ejecutable. Para el caso de latencias bajas (<35ms) ningún jugador se veía más perjudicado que otro. Para el caso opuesto, ante latencias altas (>100ms) los jugadores de ambos ejecutables se veían perjudicados por igual.

2. Precisión en los disparos: Para esta prueba se volvieron a crear dos ejecutables. En el primero de ellos, el sistema de registro de impactos no compensaba la latencia de los distintos clientes mientras que en el segundo ejecutable sí se llevaba a cabo esta compensación. Tras varias sesiones de juego de corta duración y los comentarios de los distintos participantes, se llegó a la conclusión de que la falta de compensación de la latencia en los algoritmos de registro de impactos en aquellas partidas con latencias de carácter medio o alto (>45ms) generaba una caída drástica tanto en la precisión de jugadores con todo tipo niveles de habilidad en el género de disparos en primera persona. En partidas con latencias de carácter bajo (<35ms) esta caída de la precisión en los disparos de los distintos jugadores también se veía ligeramente mermada.

3. Capacidad de un jugador de arruinar la partida: Para esta prueba se creó un único ejecutable en el que se aplicaban todas las técnicas de compensación de latencia implementadas hasta el momento de este *beta testing*. Además, se instaló el programa llamado Clumsy con la finalidad de alterar la calidad de la conexión a internet de ciertos clientes. Tras esta preparación se elegiría un único cliente de la partida para incrementarle de forma artificial sus valores de *jitter* y la latencia de los paquetes salientes. El resto de jugadores poseerían latencias de carácter medio (en el rango de 45ms y 60ms). Tras esto, se llegó a la conclusión que el resto de clientes se veían mínimamente afectados por las malas condiciones del cliente que usaba Clumsy. Estas consecuencias no eran nulas pero si mínimas y muy puntuales. Esto es gracias al sistema de predicción de *inputs* implementado en el lado del servidor.

Tras estas pruebas realizadas en el *beta testing* se llegó a la conclusión que el sistema mayormente criticado por los jugadores era el de predicción en el lado del cliente. No debido a que este sistema no fuese determinista o no predijese de forma correcta en base a los *inputs* y el estado actual, sino debido a las frecuentes predicciones de *inputs* en el lado del servidor que muchas veces no coincidían con aquellos *inputs* que habían llegado tarde. Por otra parte, el sistema menos criticado fue el de interpolación y extrapolación de entidades en el lado del cliente. Esto fue una vez más debido al sistema de predicción de *inputs* del servidor. Debido a que el servidor no esperaba hasta recibir los *inputs* del resto de clientes en caso de que estos se retrasasen, este sobrescribía el *input* retrasado con su predicción de tal forma que el resto de clientes podían seguir teniendo estados para interpolar.

Capítulo 6

Conclusiones

En esta sección de conclusiones, se reflexional sobre el Proyecto en su totalidad. Los logros obtenidos revelan el éxito en la implementación de los distintos objetivos propuestos. Las lecciones aprendidas, detallan aquellas dificultades que se han detectado a la hora de llevar a cabo este proyecto. El impacto de este proyecto manifiesta la utilidad del mismo de cara a aquellos interesados en la creación de una experiencia multijugador en línea. Además, se identifican líneas futuras y aspectos de mejora que no han sido posibles integrar en este proyecto a fecha de esta memoria.

7.1 Logros alcanzados

En este apartado se llevará a cabo un proceso de reflexión y análisis sobre el grado de cumplimiento de los objetivos propuestos al inicio del proyecto, junto las razones que corroboran dicho cumplimiento.

Uno de los objetivos principales de este proyecto fue dotar al prototipo desarrollado de un soporte multijugador en línea. Este objetivo ha sido alcanzado, puesto que el prototipo es completamente jugable tanto a través de una conexión a internet como de una conexión *loopback*. En el caso de usar una conexión a internet, los clientes se conectan al servidor a través de un *servidor de relay* que actúa como intermediario, redirigiendo los paquetes de cada uno de los *hosts* para facilitar la conexión y comunicación.

No obstante, a lo largo del desarrollo, el prototipo ha perdido la capacidad de funcionar como *host,* siendo el modo cliente y el modo servidor los únicos disponibles. Esta limitación se debe principalmente al incremento de la complejidad de los distintos sistemas del prototipo. El modo *host* no solo es una alternativa a los servidores dedicados sino que junto con una conexión *loopback* proporciona el soporte necesario para crear un modo *offline*.

Otro objetivo fundamental de este proyecto está relacionado con la calidad de la experiencia de juego por parte de los distintos jugadores. Para que esta sea lo mejor posible, es importante implementar técnicas que sean capaces de ocultar la latencia al jugador final y que sean robustas ante los problemas del *jitter* y la pérdida de paquetes, comunes en experiencias en línea, ya que estos degradan aspectos como la responsividad del juego.

En relación con este objetivo, se han implementado una gran variedad de técnicas y sistemas, capaces de hacer frente a los problemas de la red con el objetivo de minimizar el impacto que estas puedan tener en la experiencia final del juego. Entre estos sistemas se pueden destacar los siguientes:

1. Sistema de interpolación y extrapolación de entidades en el lado del cliente: Su objetivo es suavizar el movimiento del resto de jugadores ante los posibles problemas del *jitter* y la pérdida de paquetes de la red.

2. Sistema de predicción del estado local del jugador y reconciliación con el servidor: Busca mejorar la responsividad de sus acciones en el mundo virtual en base a sus *inputs*.

3. Sistema de *Hit Registration* en el lado del servidor: Tiene como objetivo minimizar los errores de precisión de los disparos de los distintos jugadores.

Por otro lado, a pesar de que estos sistemas cumplen su función correctamente, existe margen para mejorar su eficacia. Por ejemplo, como se detalla en el apartado de Líneas futuras, algunos sistemas como la extrapolación de entidades o el *Hit Registration,* podrían ser mejorados. En dicha sección, se expone el problema que poseen estas técnicas actualmente, junto con posibles soluciones para mitigarlos.

7.2 Lecciones aprendidas

Desde el inicio de este proyecto de fin de grado hasta su versión más reciente, el proceso de desarrollo se ha visto dificultado en gran medida por la falta de recursos teóricos y, sobre todo, prácticos. Gran parte de la implementación del código de red de alto nivel se ha construido desde cero, basándose únicamente en los distintos conceptos teóricos previamente investigados. Esto ha resultado en un aumento en la duración del desarrollo del prototipo FPS multijugador en línea, en comparación con la planificación prevista. Sin embargo, esta dificultad ha demostrado ser beneficiosa para el autor puesto que gracias a este esfuerzo adicional, se ha podido entender en mucha mayor profundidad la teoría aplicada. Esto se debe a que, a la hora de trasladar un concepto teórico a su implementación en el código, es necesario abordar cuidadosamente todo tipo de detalles que de otra manera podrían haber pasado desapercibidos, permitiendo así un entendimiento completo de los aspectos teóricos.

Otro aspecto importante para destacar son las pruebas unitarias. En un proyecto de gran envergadura como este, el cual involucra una variedad de sistemas diferentes, asegurar el funcionamiento correcto de cada uno de ellos es esencial y para conseguirlo, las pruebas unitarias son una buena solución. La falta de pruebas unitarias en algunos de los sistemas más importantes de este proyecto ha dificultado la resolución de pequeños errores dentro de estos sistemas.

A pesar de la escasez de pruebas unitarias en el desarrollo de este prototipo, se han llevado a cabo numerosas pruebas con jugadores reales, quienes han aportado una retroalimentación muy valiosa y han ayudado a revelar muchos errores que, o bien han estado ocultos para el

desarrollador o bien, debido a la gran cantidad de horas de trabajo invertidas, han sido normalizados.

7.3 Impacto del proyecto

Este apartado se enfoca en las diversas maneras en las que este proyecto podría influir tanto en la comunidad de desarrolladores como en aquellos interesados en explorarlo.

Este proyecto ofrece una oportunidad valiosa para comprender en profundidad cómo funcionan los distintos sistemas de un videojuego multijugador en línea, concretamente, un videojuego con una arquitectura cliente-servidor basado en la interpolación de *snapshots*. Debido a la carencia actual de un gran número de conocimientos avanzados que indaguen en los ámbitos del *netcode* y el multijugador, la teoría plasmada en los apartados de Evolución histórica de los videojuegos multijugador y Complementos teóricos representan un recurso completo y sólido para aquellos que se están iniciando y desean ampliar sus conocimientos.

Por otra parte, este proyecto no solo puede resultar beneficioso para aquellos en el proceso de iniciación, sino también puede servir como una fuente de inspiración técnica para todos esos profesionales que ya disponen de conocimientos en el tema y buscan una comprensión más detallada sobre la implementación. En este sentido, el apartado de Desarrollo aborda de manera exhaustiva esta necesidad.

7.4 Líneas futuras

A lo largo de este apartado, se van a detallar las posibles futuras implementaciones que no han podido llevarse a cabo en la entrega actual pero que sin duda mejorarían algún aspecto multijugador.

Delta snapshots

Las delta *snapshots* son actualizaciones parciales del estado de juego [69, 70]. Cuando un cliente se conecta al servidor, este le enviará la totalidad del estado de juego. Una vez llegue al cliente este enviará al servidor el identificador de dicha *snapshot* para que el servidor sepa qué información posee el jugador. De esta forma, las siguientes actualizaciones del estado de juego pueden ser creadas únicamente reflejando los cambios respecto a la última actualización confirmada por el cliente. A esto se le conoce como *delta snapshot* y es una técnica de compresión para reducir el tamaño de las actualizaciones del estado de juego transmitidas a través de la red y como consecuencia el ancho de banda. El Quake III hace uso de estas delta *snapshots* [16] y en la web de Fabien [71] se puede encontrar un ejemplo explicado paso a paso de su implementación

Mejora del sistema de Hit Registration

De cara al futuro una mejora en el prototipo sería el uso de algoritmos más avanzados dedicados al registro de impactos como el algoritmo usado en Valorant o incluso el usado en Overwatch explicado en el apartado de Registro de impactos.

Añadir algoritmo de Interest Management

Con el objetivo de reducir aún más el ancho de banda y el tamaño de las actualizaciones de estado, se ha pensado implementar un algoritmo de gestión de la relevancia para que el servidor únicamente transmita aquellas entidades relevantes para el cliente destinatario. Además, esto puede suponer una mejora en la seguridad del juego.

Mejora del sistema de Jitter buffer de inputs del servidor

Otra mejora sería reducir la tolerancia del servidor a no recibir ningún *input* de un cliente en específico. Si se consigue reducir el umbral de veces consecutivas que el Buffer de jitter ha estado vacío, el servidor predeciría el *input* perdido en cuanto el *buffer* se quede vacío por primera vez. Esto haría que siempre se tuvieran *inputs* para simular independientemente de si son llegados por parte del cliente o predichos por el servidor. Aunque, como desventaja, el reducir esta tolerancia podría aumentar el número de predicciones de *inputs* y, como consecuencia, el número de posibles reconciliaciones en el cliente afectado. Por otra parte, esto supondría una ventaja ya que los jugadores con buena conexión no se verían perjudicados debido a los jugadores con mala conexión. Actualmente, si no llegan *inputs* al servidor, este no avanzará el estado de dicho jugador y por lo tanto tampoco se avanzaría en el resto de clientes (Puesto que se generaría el mismo estado de entidad). Pero si tras este periodo de ausencia de *inputs* llegan todos los que tendrían que haber llegado con anterioridad, muchos de ellos seguirán siendo válidos y al superar el umbral del tamaño máximo del *buffer*, estos se procesarían en el mismo *tick* generando inconsistencias al interpolar en el resto de clientes. Actualmente estas inconsistencias son mínimas, pero no nulas. De ahí de la mejora de este sistema propuesta para el futuro.

Mejora del sistema de extrapolación

Como se ha comentado anteriormente, el prototipo usa el algoritmo de mezcla de velocidades proyectadas para llevar a cabo la extrapolación. Para el futuro, se quieren probar distintos algoritmos para así comparar los resultados y determinar cual se ajusta mejor en este caso concreto. Por otra parte, se quiere añadir un número máximo de veces consecutivas que el algoritmo de extrapolación puede estar activo. Se ha demostrado que cuanto más tiempo estén los algoritmos de extrapolación prediciendo futuros estados, mayor podrá el error en la diferencia con el estado actual. Es por ello que si se supera un cierto umbral, la entidad deje de extrapolar y adopte otro comportamiento (p.ej. quedarse parada)

Creación de un sistema de envío de inputs de tick variable

Actualmente los clientes generan *inputs* a la misma velocidad que el servidor genera *snapshots*, es decir, 50 veces por segundo. Pero si el cliente es capaz de generar *inputs* de tal forma que estos estén sincronizados con la tasa de renderizado, se puede conseguir reducir el *input lag* [70]. Para ello, cada *input*, deberá de guardar el instante de tiempo en el que fue generado. De esta forma, se podrá calcular el tiempo transcurrido entre dos *inputs* consecutivos para avanzar la simulación. Esta técnica presenta ciertas desventajas. Una de ellas puede suponer un impacto en el rendimiento, tanto del cliente como del servidor, en caso de que el *tick* variable tome valores muy altos. Esto es debido al número de simulaciones que deben realizarse. Para ello, se limita un mínimo y un máximo permitido. Otro punto negativo que se comenta en el artículo de SnapNet es que jugadores con distintas tasas de generación de *inputs* pueden tomar trayectorias distintas. Por ejemplo, en una mecánica de

salto, donde el jugador realiza un movimiento parabólico, cuanto mayor sea la tasa de *inputs* del jugador, mayor será la aproximación del movimiento al arco parabólico resultante. Si un jugador tiene una tasa de *inputs* de 120Hz, su arco parabólico estará formado por 120 segmentos. En cambio, un jugador con una tasa de *inputs* de 30Hz se aproximará menos al arco parabólico ideal ya que este movimiento estará formado por 30 segmentos 4 veces mayores al del jugador con una tasa de 120Hz. Otra desventaja que puede suponer las altas tasas de *inputs* es un impacto negativo en el rendimiento, tanto del cliente como del servidor. Esto se podría solucionar combinando varios *inputs* y avanzando la simulación una única vez a partir de un *input* más grande aunque esta técnica es susceptible a la primera desventaja comentada. Esto es debido al número de simulaciones que deben realizarse. Por estas dos razones, es recomendable fijar una tasa mínima y un máxima de *inputs* permitida.

Bibliografía

[1] W. Higginbotham, «Tennis For Two,» [En línea]. Available: https://es.wikipedia.org/wiki/Tennis_for_Two.

[2] «Spacewar!,» [En línea]. Available: https://es.wikipedia.org/wiki/Spacewar!.

[3] Atari, «Pong,» [En línea]. Available: https://es.wikipedia.org/wiki/Pong.

[4] «PLATO,» [En línea]. Available: https://es.wikipedia.org/wiki/Programmed_Logic_Automated_Teaching_Operations.

[5] «MUD,» [En línea]. Available: https://es.wikipedia.org/wiki/MUD_(videojuegos).

[6] «DOOM,» [En línea]. Available: https://es.wikipedia.org/wiki/Doom_(videojuego_de_1993).

[7] «Quake,» [En línea]. Available: https://es.wikipedia.org/wiki/Quake.

[8] J. v. Waveren, «The DOOM III Network Architecture,» 2006.

[9] «QuakeWorld,» [En línea]. Available: https://es.wikipedia.org/wiki/QuakeWorld.

[10] «Log de John Carmack,» [En línea]. Available: https://fabiensanglard.net/quakeSource/johnc-log.aug.htm.

[11] «Duke Nukem 3D,» 29 Enero 1996. [En línea]. Available: https://en.wikipedia.org/wiki/Duke_Nukem_3D. [Último acceso: 17 Abril 2023].

[12] «Build Engine Source Code,» [En línea]. Available: https://github.com/fabiensanglard/vanilla_duke3D.

[13] «Entrevista a Ken Silverman,» [En línea]. Available: https://www.pressreader.com/uk/retro-gamer/20180614/281565176453517.

[14] M. Frohnmayer y T. Gift, «The TRIBES Engine Networking Model,» 2000.

[15] J. Glazer y S. Madhav, Multiplayer Game Programming, Adison-Wesley Professional, 2015.

[16] D. Stefyn, A. Cricenti y P. Branch, «Quake III Arena Game Structures,» 2011.

[17] «Huffman Coding,» [En línea]. Available: https://en.wikipedia.org/wiki/Huffman_coding.

[18] «Quake III Arena GitHub Repository,» [En línea]. Available: https://github.com/id-Software/Quake-III-Arena.

[19] M. Trentacoste, «A Survey Of Visibility Methods For Networked Games».

[20] J. Mattis, «Performing Lag Compensation in Unreal Engine 5,» SnapNet, 21 Febrero 2023. [En línea]. Available: https://www.snapnet.dev/blog/performing-lag-compensation-in-unreal-engine-5/.

[21] G. Fiedler, «Snapshot Compression,» Gaffer On Games, 4 Enero 2015. [En línea]. Available: https://gafferongames.com/post/snapshot_compression/.

[22] Y. Bernier, «Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization,» Valve, 2001. [En línea]. Available: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization.

[23] M. A. Herrmann, «Automated Network Compression For Online Games,» 2016.

[24] «IP fragmentation,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/IP_fragmentation.

[25] «Run-Length Encoding,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Run-length_encoding.

[26] J. Henry, «Back to the Future! Working with Deterministic Simulation in 'For Honor',» GDCVault, Marzo 2019. [En línea]. Available: https://www.gdcvault.com/play/1026077/Back-to-the-Future-Working.

[27] P. Bettner y M. Terrano, «Netowrk Programming in Age Of Empires and Beyond,» 2001.

[28] «Havok,» [En línea]. Available: https://www.havok.com.

[29] J. Saldana y M. Suznjevic, «QoE and Latency Issues in Networked Games,» 2015.

[30] A. Seetharam, «Network Delay - Transmission and Propagation Delay,» [En línea]. Available: https://youtu.be/zUPayNytbgw.

[31] CodingPanelEditor, «Delays in Computer Networks and Communication,» 10 Marzo 2021. [En línea]. Available: https://www.codingpanel.com/delays-in-computer-networks-and-communication-formula-with-examples-transmission-delay-propagation-delay-processing-delay-queuing-delay/.

[32] G. Fiedler, «Packet Fragmentation and Reassembly,» 6 Septiembre 2016. [En línea]. Available: https://gafferongames.com/post/packet_fragmentation_and_reassembly/.

[33] R. Walsh, «What is IP Fragmentation?,» 20 Marzo 2023. [En línea]. Available: https://www.comparitech.com/blog/information-security/what-is-ip-fragmentation/.

[34] «Bufferbloat,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Bufferbloat.

[35] «Active Queue Management,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Active_queue_management.

[36] T. Engel, «Finding the right Network Topology for your Multiplayer Game,» 19 Agosto 2022. [En línea]. Available: https://www.youtube.com/watch?v=IWi6iv7_sT4.

[37] J. Lindström y T. Malm, «Development of a real-time multiplayer game for the computer tablet,» 2012.

[38] I. Dubrovin, «Shared World Illusion in Networked Videogames,» 2016.

[39] F. Dang Tran, M. Deslaugiers, A. Gérodolle, L. Hazard y N. Rivierre, «An Open Middleware for Large-scale Networked Virtual Environments,» 2002.

[40] T. A. Funkhouser, «RING: A Client-Server System for Multi-User Virtual Environments,» 1995.

[41] «Remote procedure call,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Remote_procedure_call.

[42] «About Netcode for GameObjects,» [En línea]. Available: https://docs-multiplayer.unity3d.com/netcode/current/about/.

[43] «Photon Engine,» [En línea]. Available: https://www.photonengine.com.

[44] M. Claypool, S. Liu y X. Xu, «A Survey And Taxonomy Of Latency Compensation Techniques For Network Computer Games,» Worcester Polytechnic Institute, 2021.

[45] M. Claypool y S. Liu, «L33t Or N00b? How Player Skill Alters The Effects Of Network Latency On First Person Shooter Game Players,» Worcester Plytechnic Institute, 2021.

[46] P. Quax, P. Monsieurs, W. Lamotte, D. De Vleeschauwer y N. Degrande, «Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game,» 2004.

[47] R. Vicencio-Moreira, R. L. Mandryk, C. Gutwin y S. Bateman, «The Effectiveness (or Lack Thereof) of Aim-Assist Techniques in First-Person Shooter Games,» 2014.

[48] G. Gambetta, «Fast-Paced Multiplayer (Part III): Entity Interpolation,» [En línea]. Available: https://www.gabrielgambetta.com/entity-interpolation.html.

[49] «Source Multiplayer Networking,» [En línea]. Available: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

[50] F. Stakem y G. AlRegib, «An Adaptative Approach to Exponential Smoothing for CVE State Prediction,» 2009.

[51] T. Walker, «Dead Reckoning for Distributed Network Online Games,» 2021.

[52] P. Bourke, «Interpolation methods,» Diciembre 1999. [En línea]. Available: http://paulbourke.net/miscellaneous/interpolation/.

[53] C. Murphy, «Believable Dead Reckoning for Networked Games,» de *Game Engine Gems 2*, 2011.

[54] M. Delbosc, «Replicating Chaos, Vehicle Replication in Watch Dogs 2,» 2017.

[55] J. Lundgren, «Implementation and Evaluation of Hit Registration in Networked First Person Shooters,» 2021.

[56] M. Reid, «VALORANT's foundation is Unreal Engine,» 17 Junio 2020. [En línea]. Available: https://www.unrealengine.com/en-US/tech-blog/valorant-s-foundation-is-unreal-engine.

[57] T. Ford, «Overwatch Gameplay Architecture and Netcode,» 2017. [En línea]. Available: https://youtu.be/zrIY0elyqmI.

[58] S. W. K. Lee y R. K. C. Chang, «On "Shot Around a Corner" in First-Person Shooter Games,» 2017.

[59] Battle(non)sense, «CoD Infinite Warfare & Modern Warfare Netcode Analysis,» https://www.youtube.com/watch?v=oKE_eaTb1TU, 2016.

[60] S. W. K. Lee y R. K. C. Chang, «Enhancing the Experience of Multiplayer Shooter Games via Advanced Lag Compensation,» 2018.

[61] M. deWet y D. Straily, «Peeking into Valorant's Netcode,» 2020.

[62] P. Haile y M. Sanborn, «Call of Duty: Modern Warfare Netcode Explained!,» 25 Octubre 2019. [En línea]. Available: https://www.youtube.com/watch?v=tCpYV4k_izE.

[63] «MMOG. RTT, Input Lag, and How to Mitigate Them,» IT Hare, 2016 Enero 25. [En línea]. Available: http://ithare.com/mmog-rtt-input-lag-and-how-to-mitigate-them/.

[64] J.-S. Boulanger, J. Kienzle y C. Verbrugge, «Comparing Interest Management Algorithms For Massively Multiplayer Games,» 2006.

[65] «Observer Pattern,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Observer_pattern.

[66] S. Benford y L. Fahlén, «A Spatial Model of Interaction in Large Virtual Environments,» 1993.

[67] C. Greenhalgh, «Awareness-Based Communication Management in the MASSIVE Systems,» 1998.

[68] «Facade pattern,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Facade_pattern.

[69] N. Geretti, «Netcode Series Part 2: Data Channels (Snapshots and RPCs),» 17 Agosto 2021. [En línea]. Available: https://medium.com/@geretti/netcode-series-part-2-data-channels-c12e9a238800.

[70] J. Mattis, «Netcode Architectures Part 3: Snapshot Interpolation,» SnapNet, 7 Junio 2023. [En línea]. Available: https://snapnet.dev/blog/netcode-architectures-part-3-snapshot-interpolation/.

[71] F. Sanglard, «QUAKE 3 SOURCE CODE REVIEW: NETWORK MODEL (PART 3 OF 5),» 30 Junio 2012. [En línea]. Available: https://fabiensanglard.net/quake3/network.php.

[72] J. Schreier, Blood, Sweat and Pixels, HarperCollins, 2017.

[73] M. Barton, Honoring the Code, CRC Press, 2016.

[74] 24 09 2019. [En línea]. Available: https://www.dell.com/es-es/shop/mochila-compacta-dell-pro-15/apd/460-bcmj/maletines-de-transporte.

Ludografía

Tennis For Two, William Higginbotham 1958.

Spacewar!, Steve Russell, 1961.

Pong, Atari, 1972.

DOOM, Id Software, 1993.

Quake, Id Software, 1996.

Quake III, Id Software, 1999.

DOOM III, Id Software, Activision, 2004.

Starsiege Tribes, Dynamix, Sierra Entertainment, 1998.

Duke Nukem 3D, 3D Realms, 1996.

Age of Empires, Ensemble Studios, Microsoft, 1997.

For Honor, Ubisoft, 2017.

Valorant, Riot Games, 2020.

Overwatch, Blizzard Entertainment, 2016

Battlefield 4, DICE, Electronic Arts, 2013

Call Of Duty: Infinite Warfare, Infinity Ward, Activision, 2016

Anexo I

Gráficas de la evaluación de los algoritmos de interpolación y extrapolación

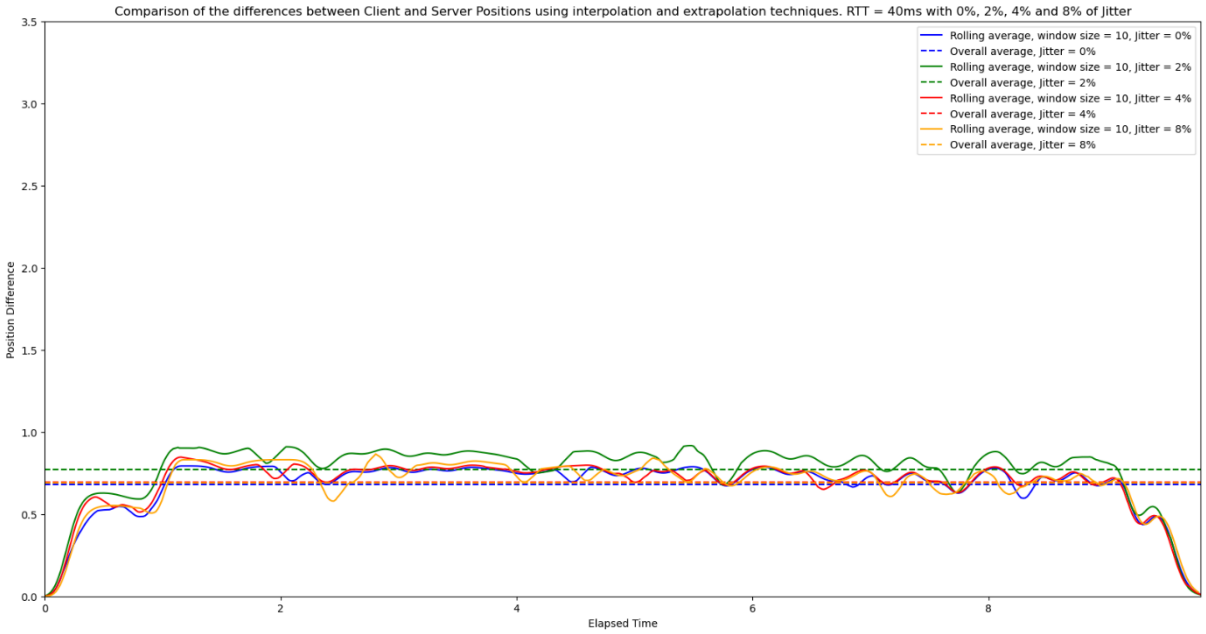


Ilustración 69 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 40 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

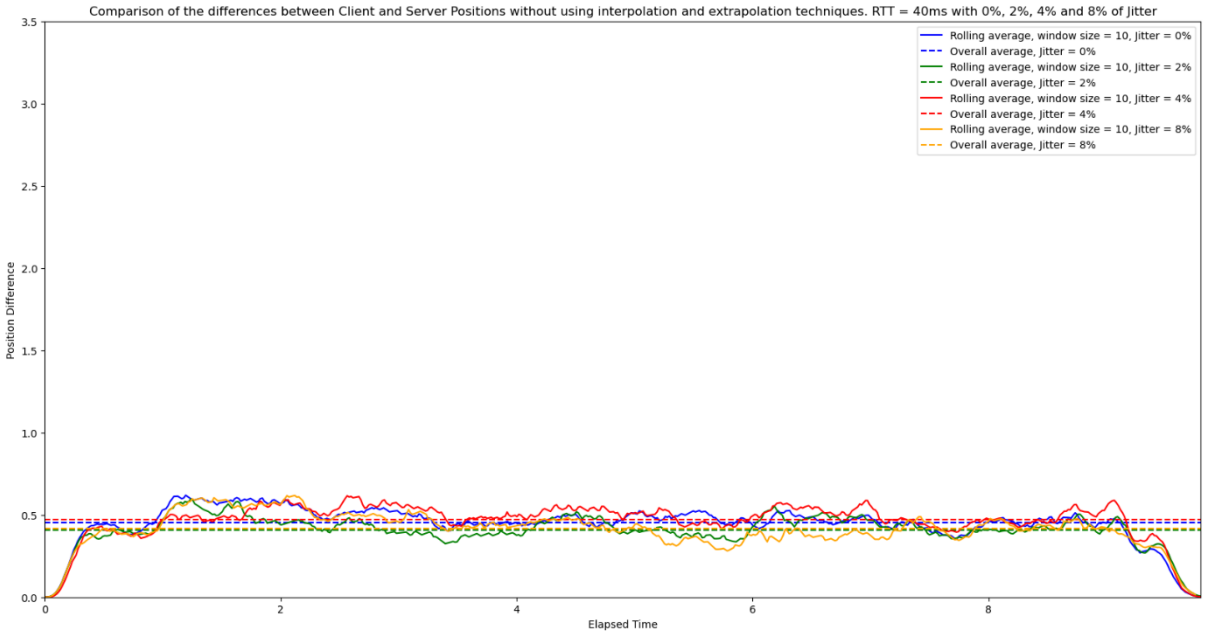


Ilustración 70 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 40 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

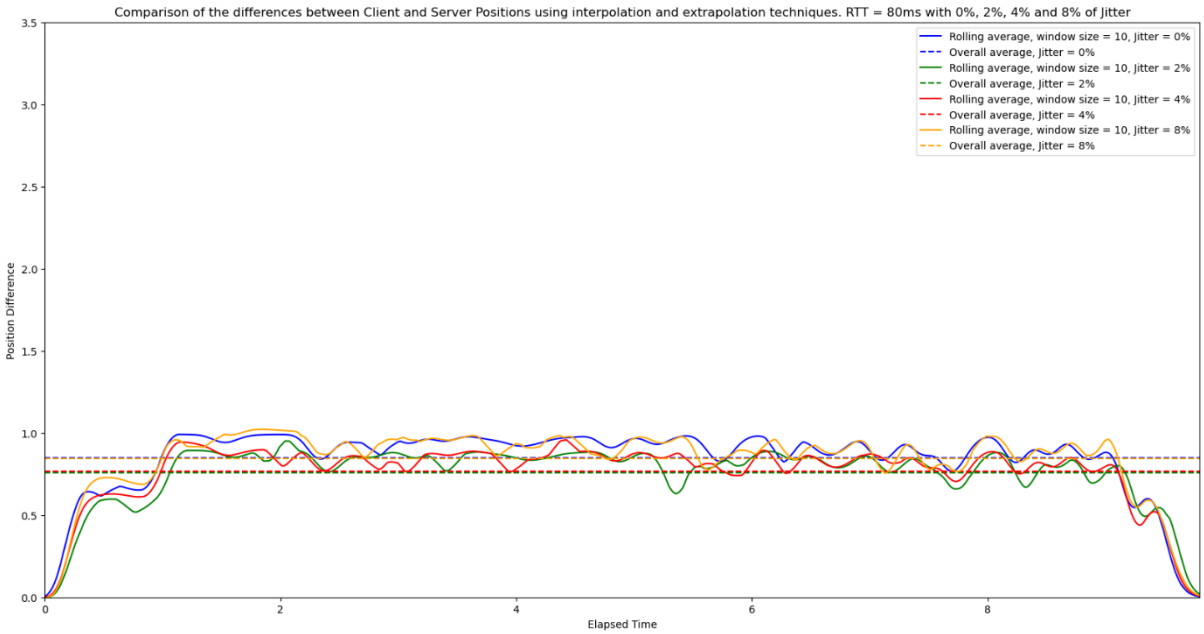


Ilustración 71 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 80 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

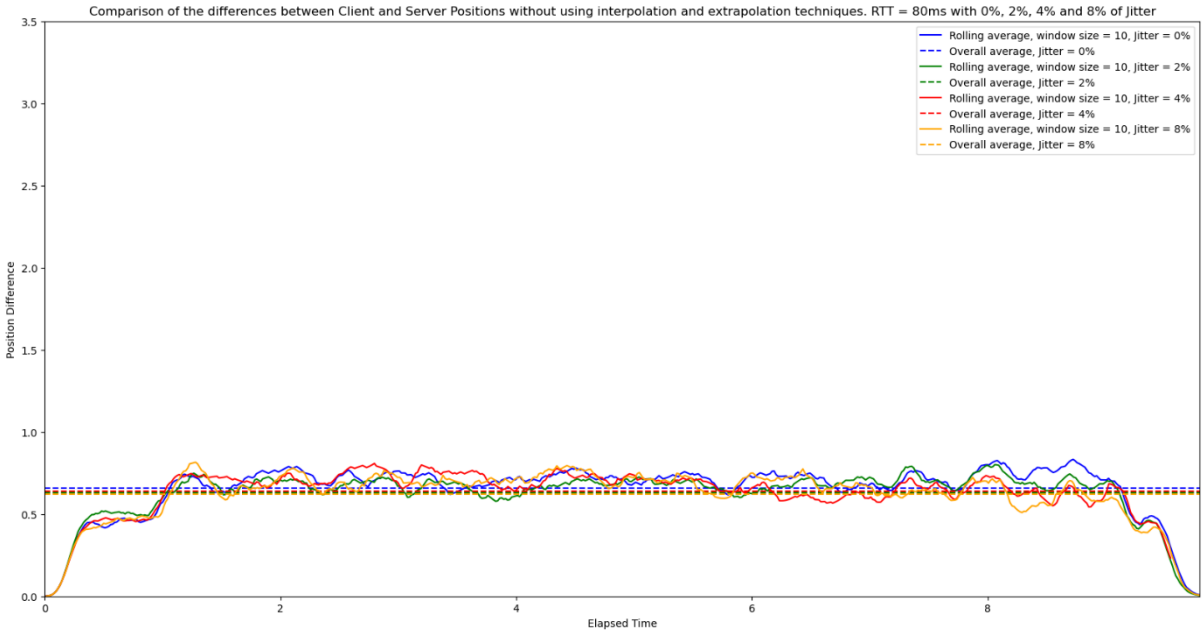


Ilustración 72 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 80 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

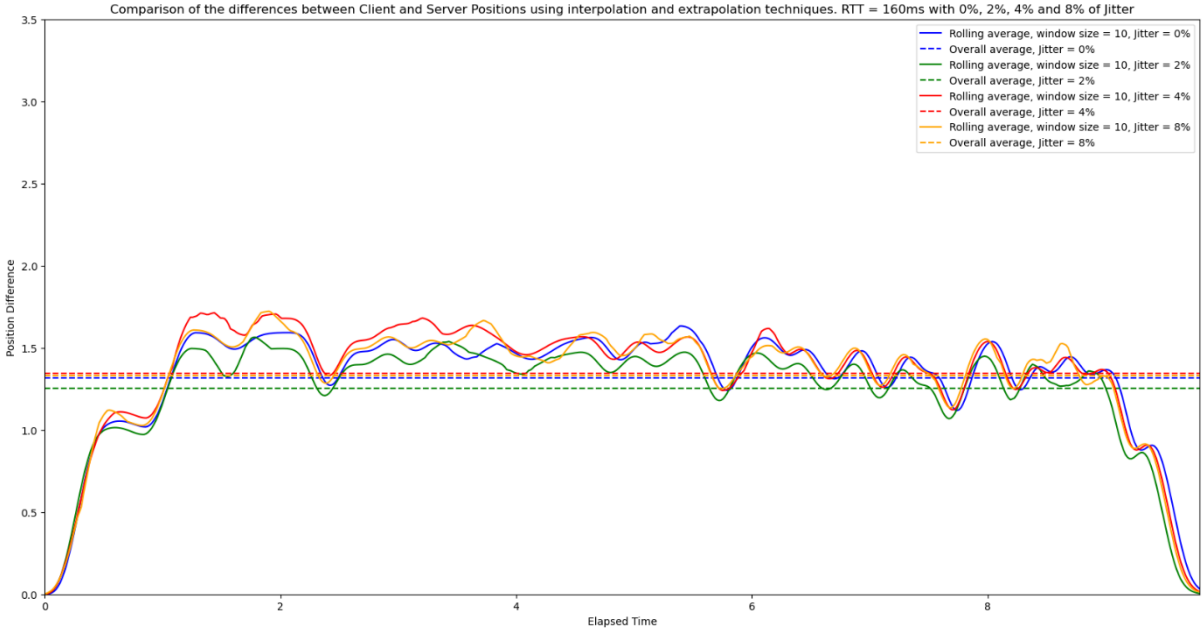


Ilustración 73 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 160 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

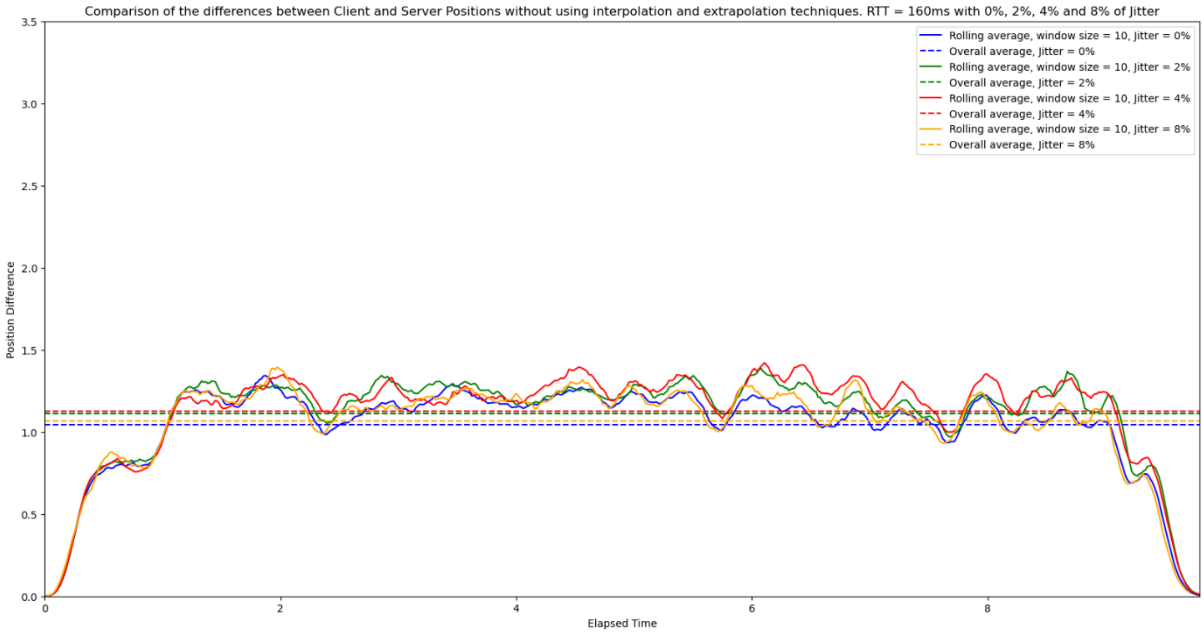


Ilustración 74 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 160 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

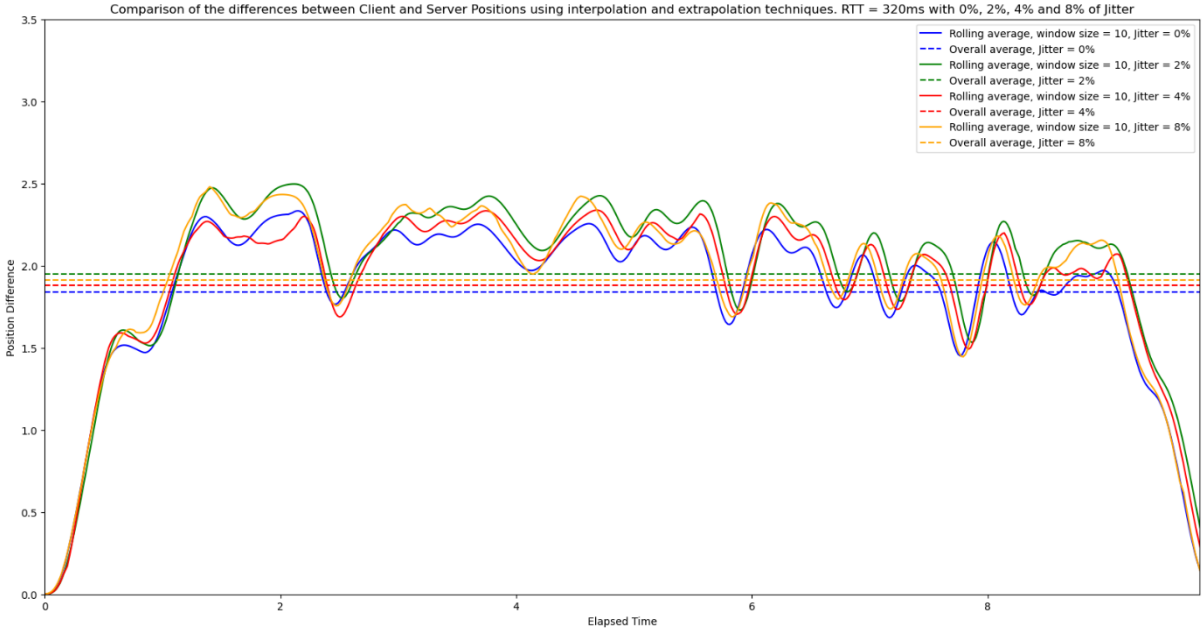


Ilustración 75 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 320 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

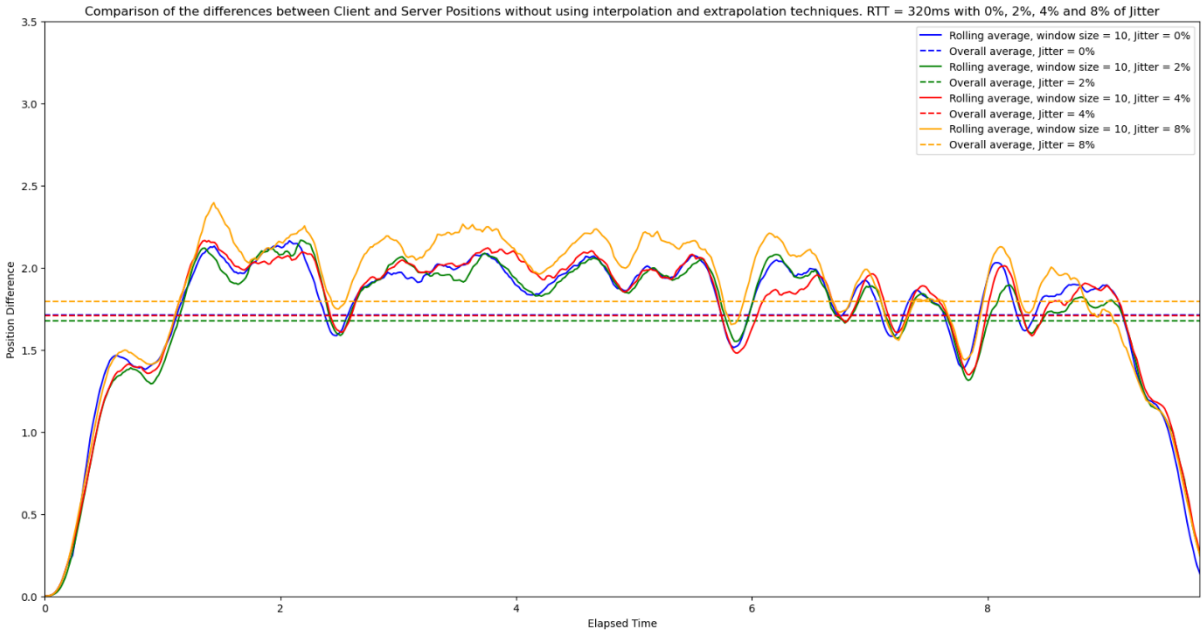


Ilustración 76 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 320 milisegundos y distintos porcentajes de jitter (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

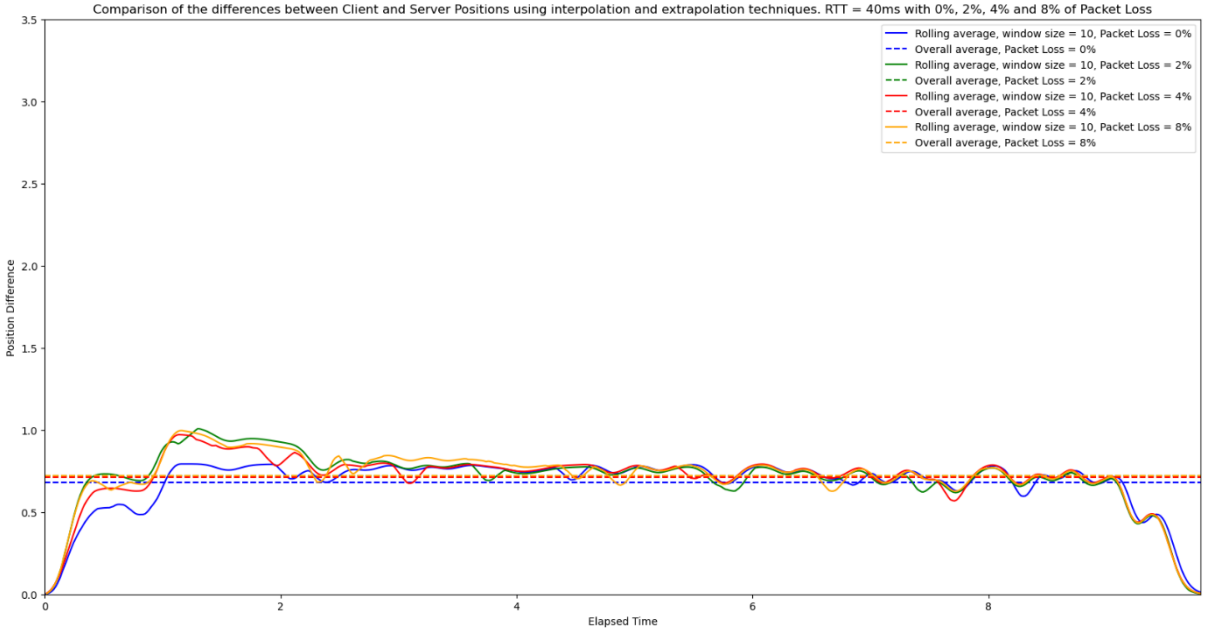


Ilustración 77 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 40 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

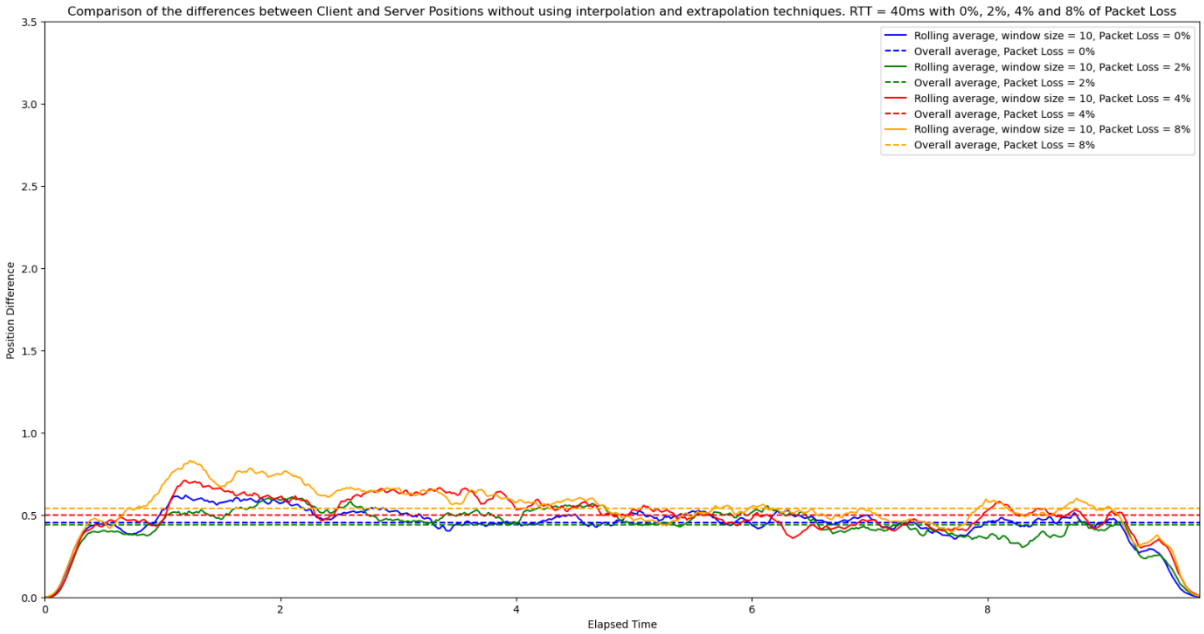


Ilustración 78 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 40 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

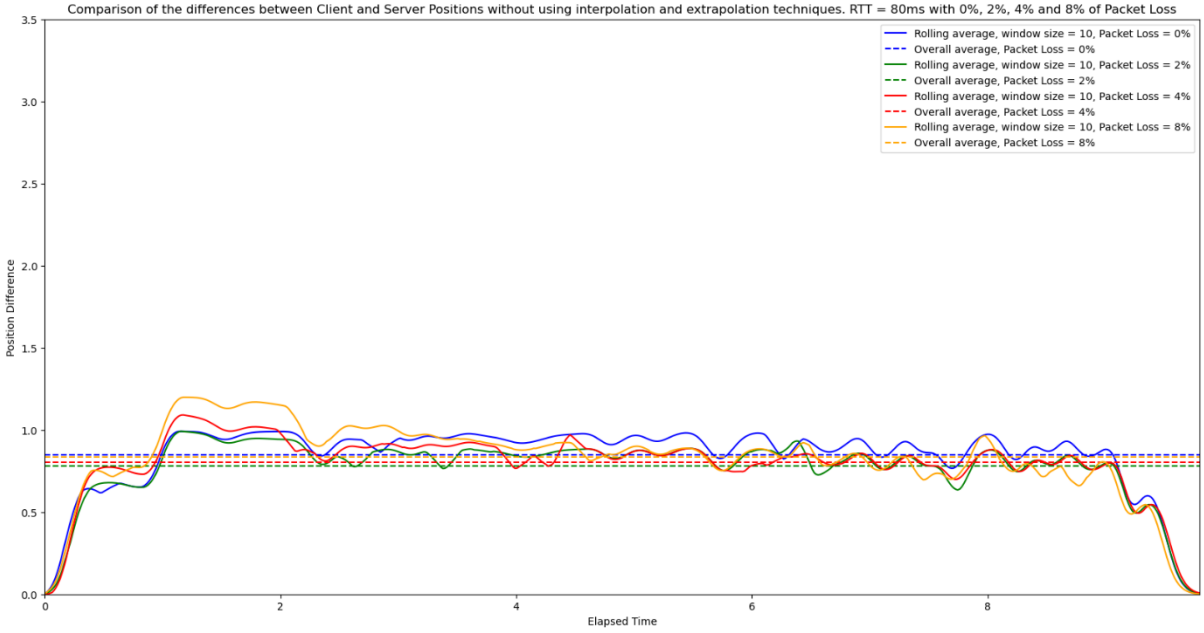


Ilustración 79 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 80 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

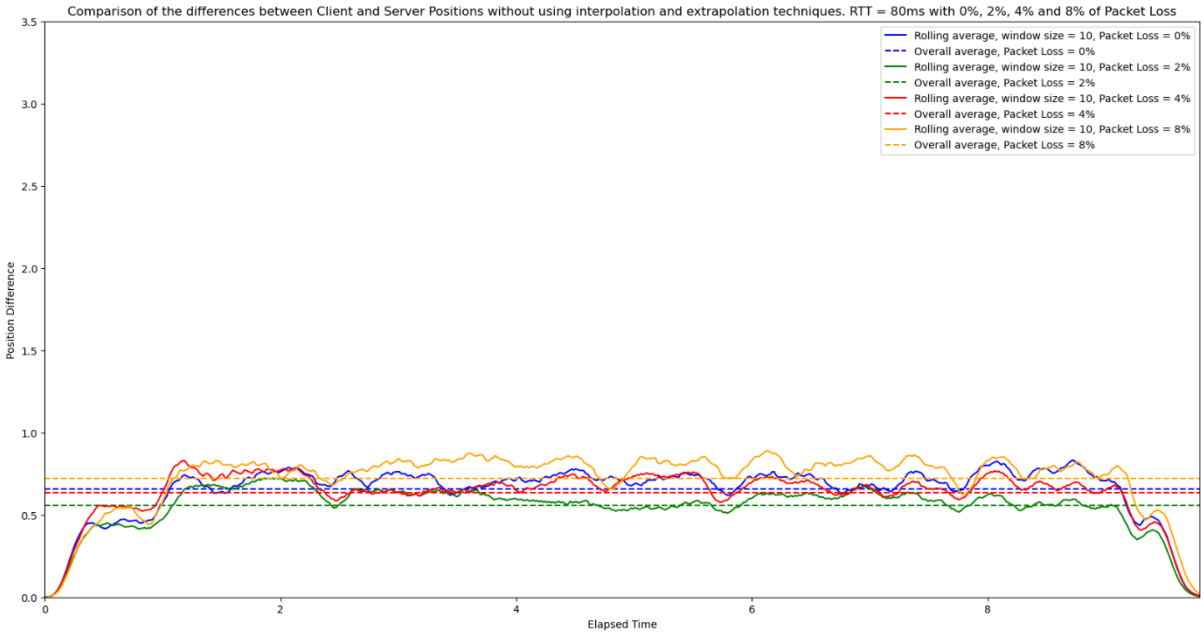


Ilustración 80 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 80 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

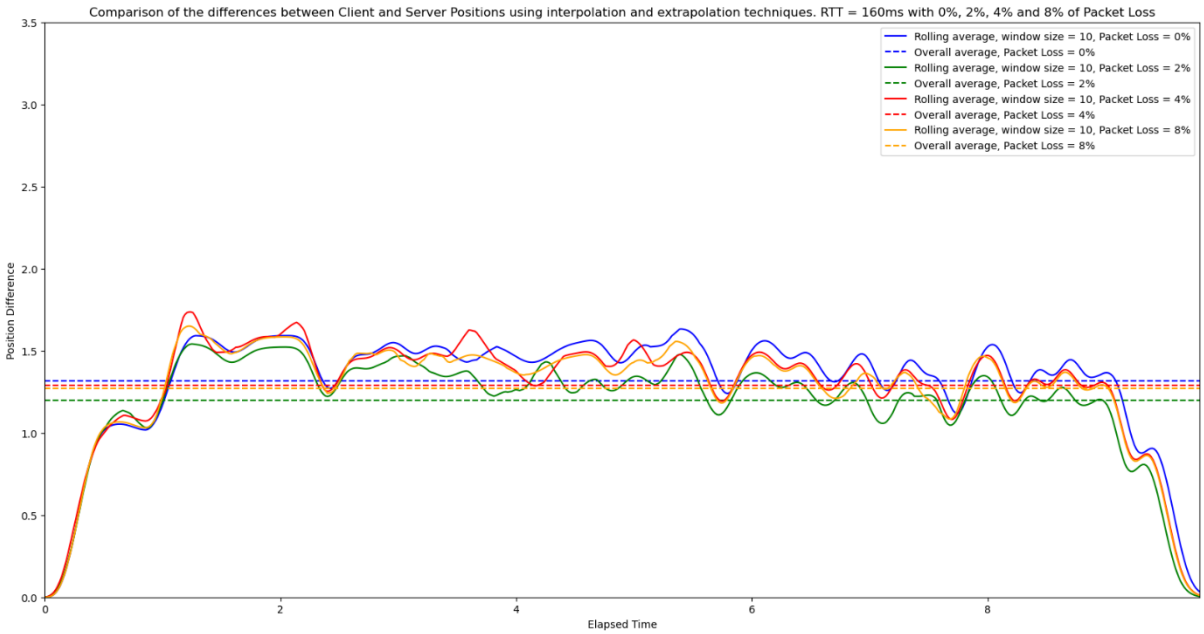


Ilustración 81 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 160 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

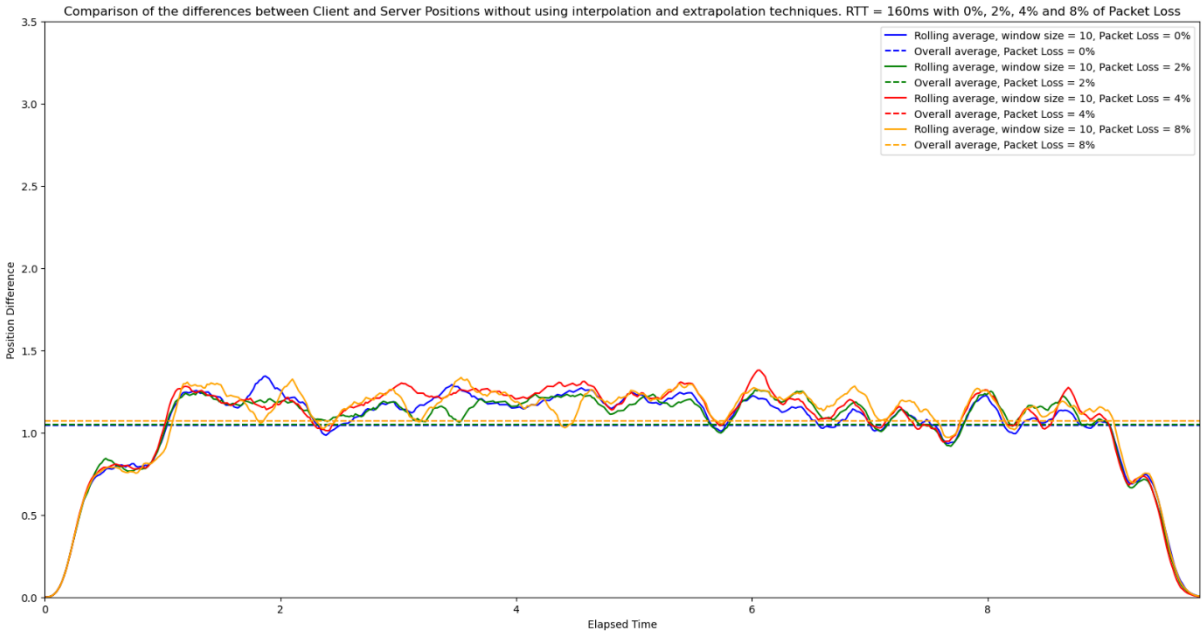


Ilustración 82 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 160 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

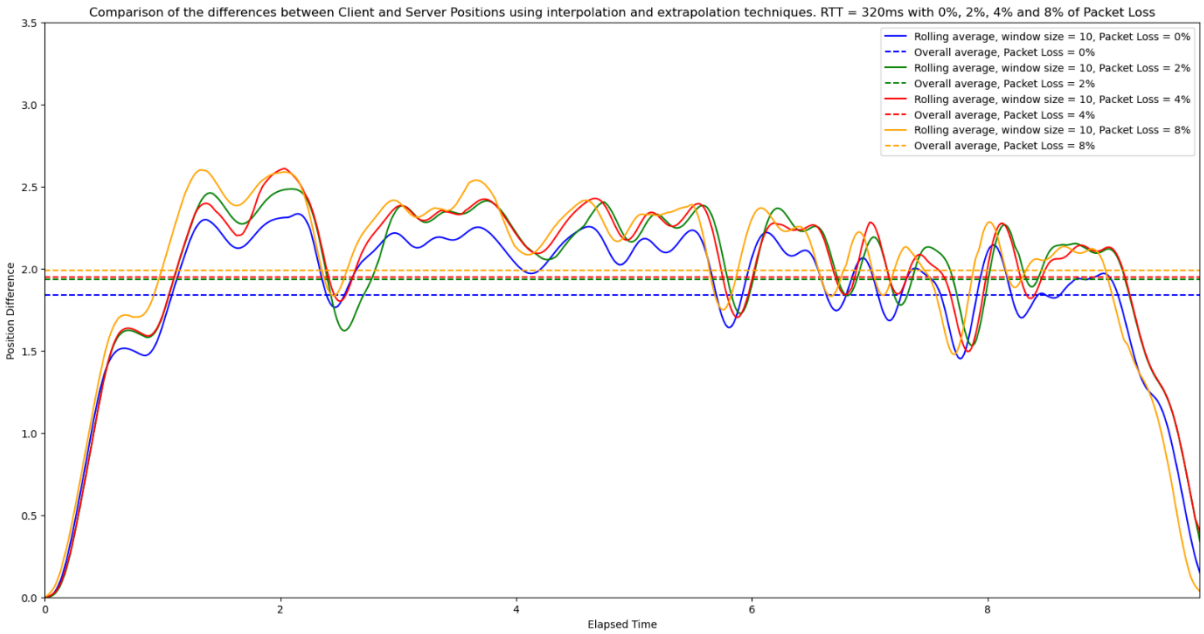


Ilustración 83 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 320 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

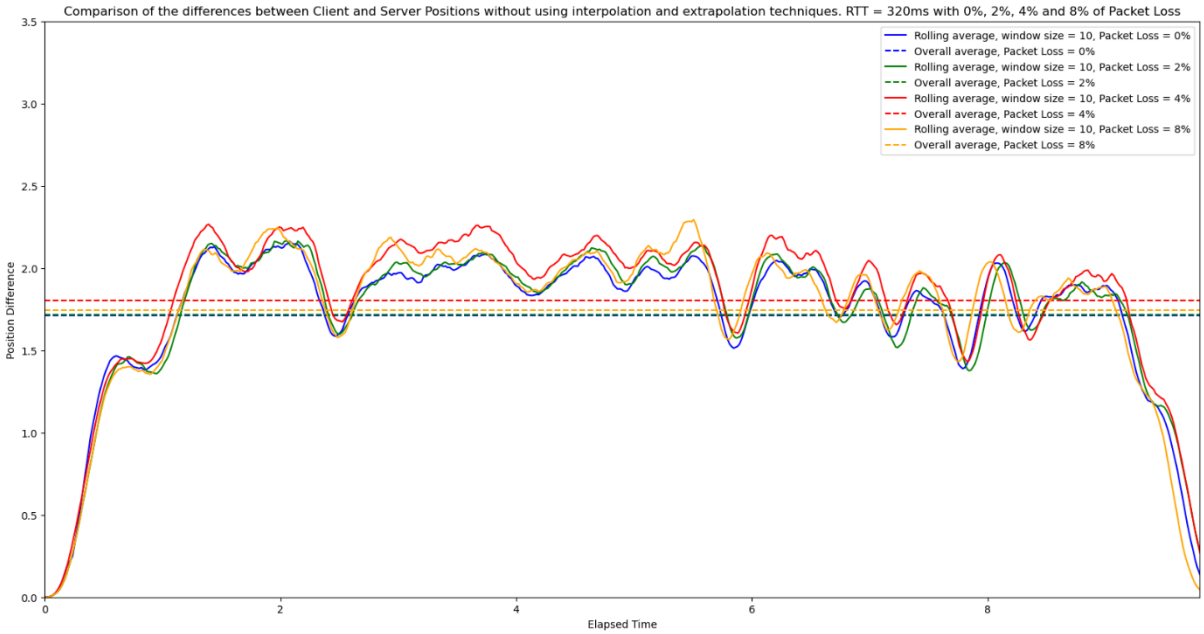


Ilustración 84 - Representación de la media global y móvil de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente con un RTT de 320 milisegundos y distintos porcentajes de pérdida de paquetes (Azul = 0%, verde = 2%, rojo = 4% y naranja = 8%).

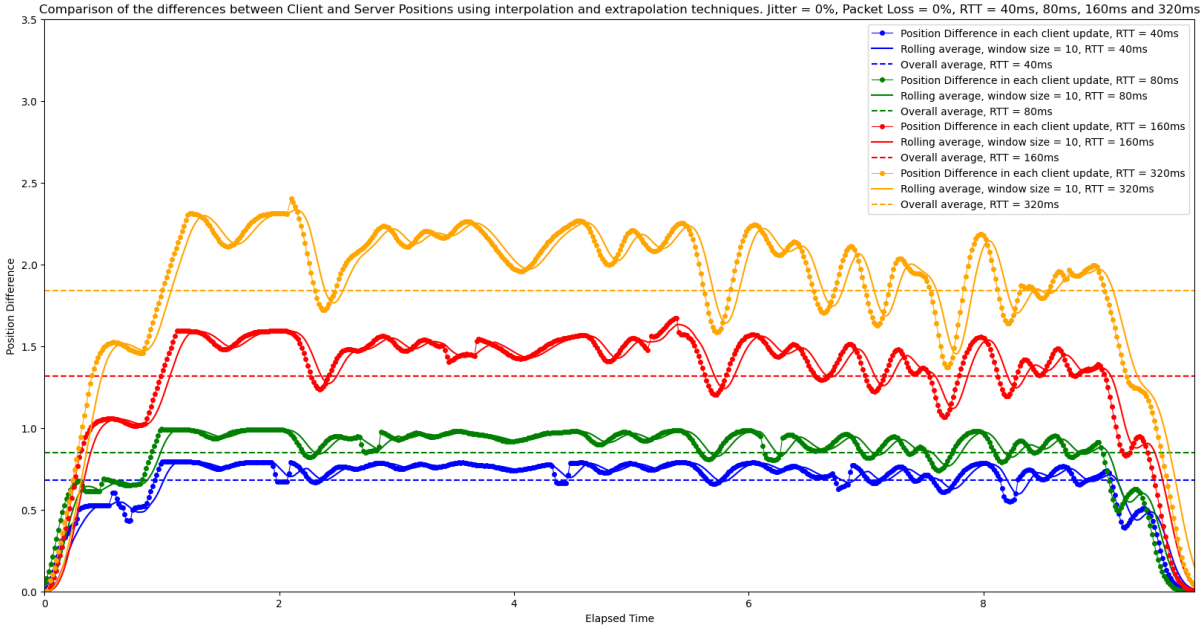


Ilustración 85 - Comparación de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 0% de jitter y 0% de pérdida de paquetes.

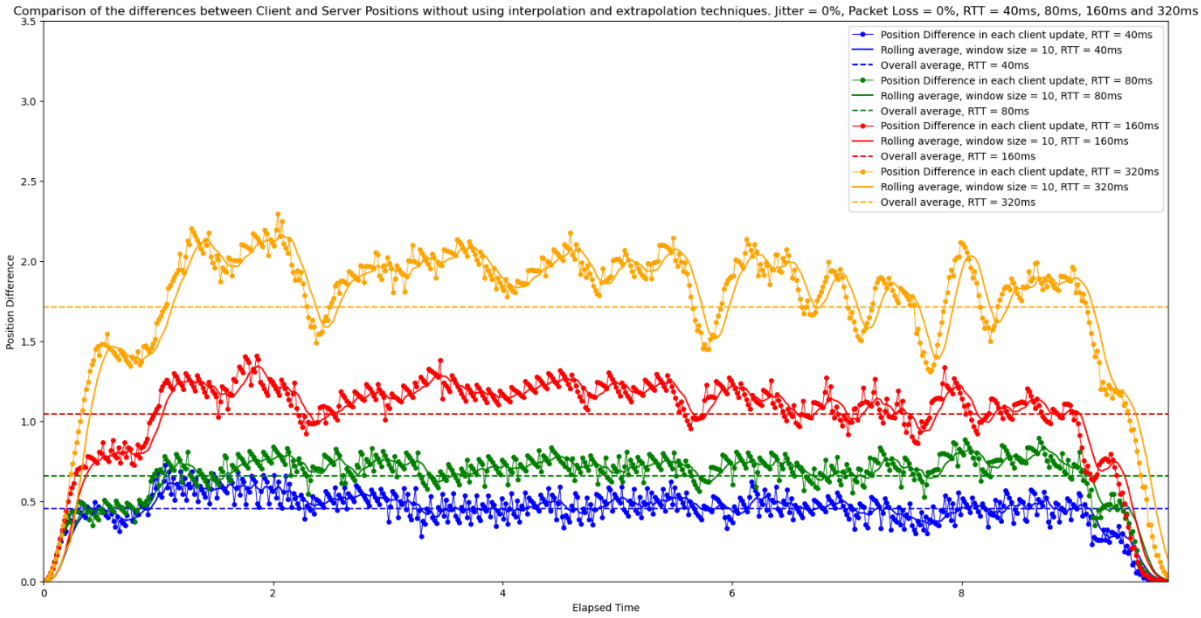


Ilustración 86 - Comparación de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 0% de jitter y 0% de pérdida de paquetes.

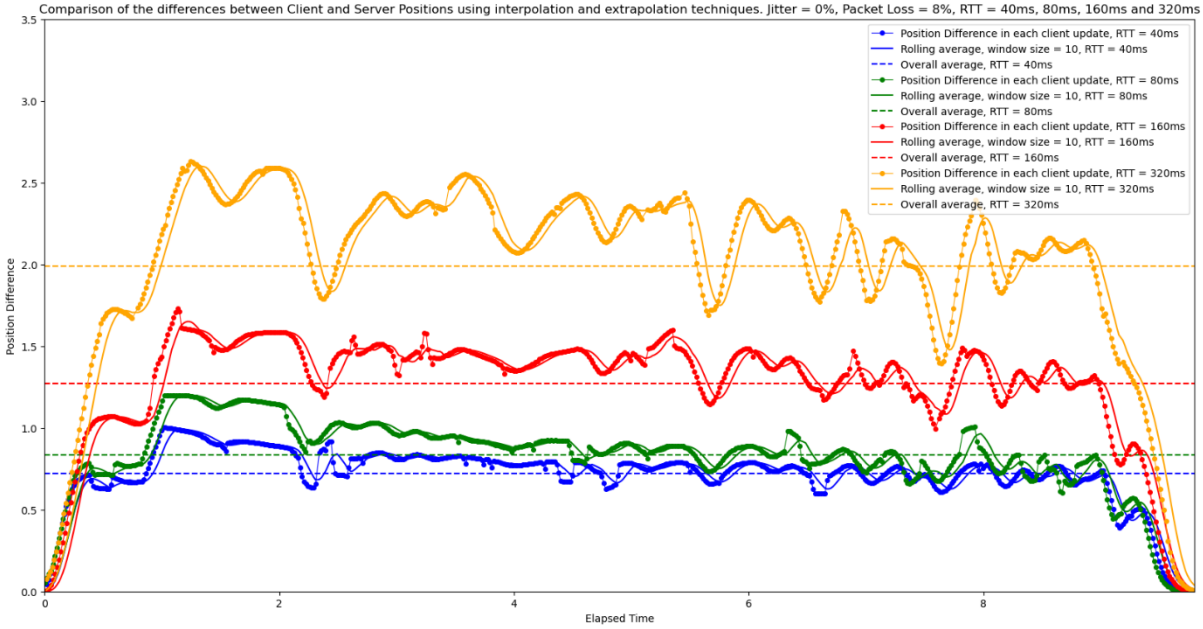


Ilustración 87 - Comparación de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 0% de jitter y 8% de pérdida de paquetes.

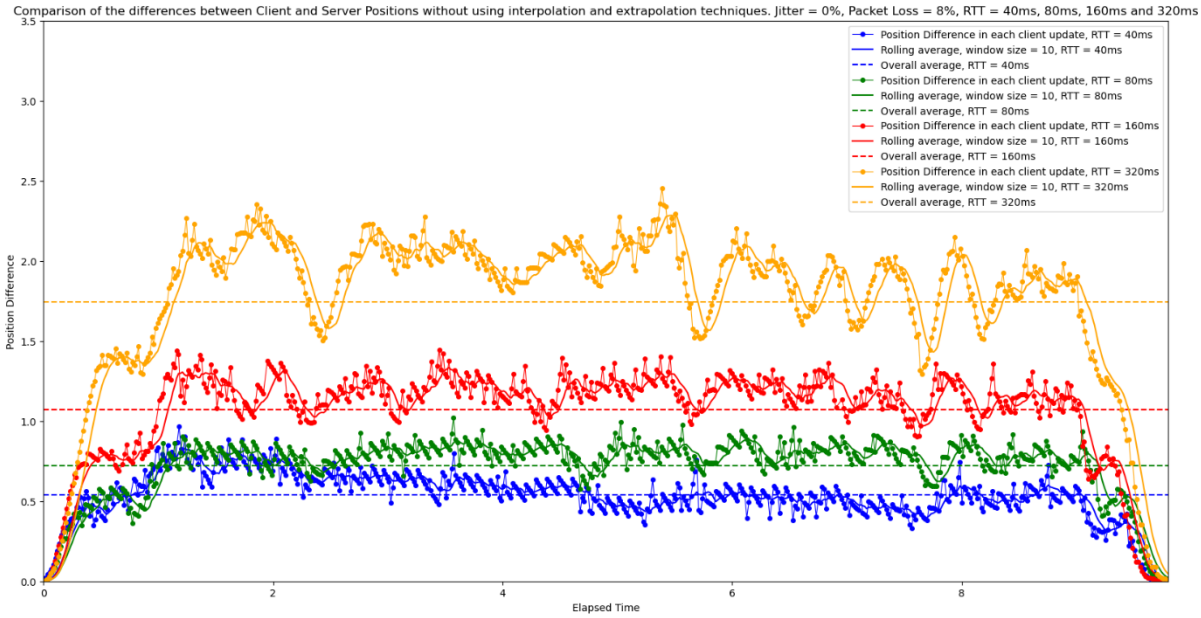


Ilustración 88 - Comparación de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 0% de jitter y 8% de pérdida de paquetes.

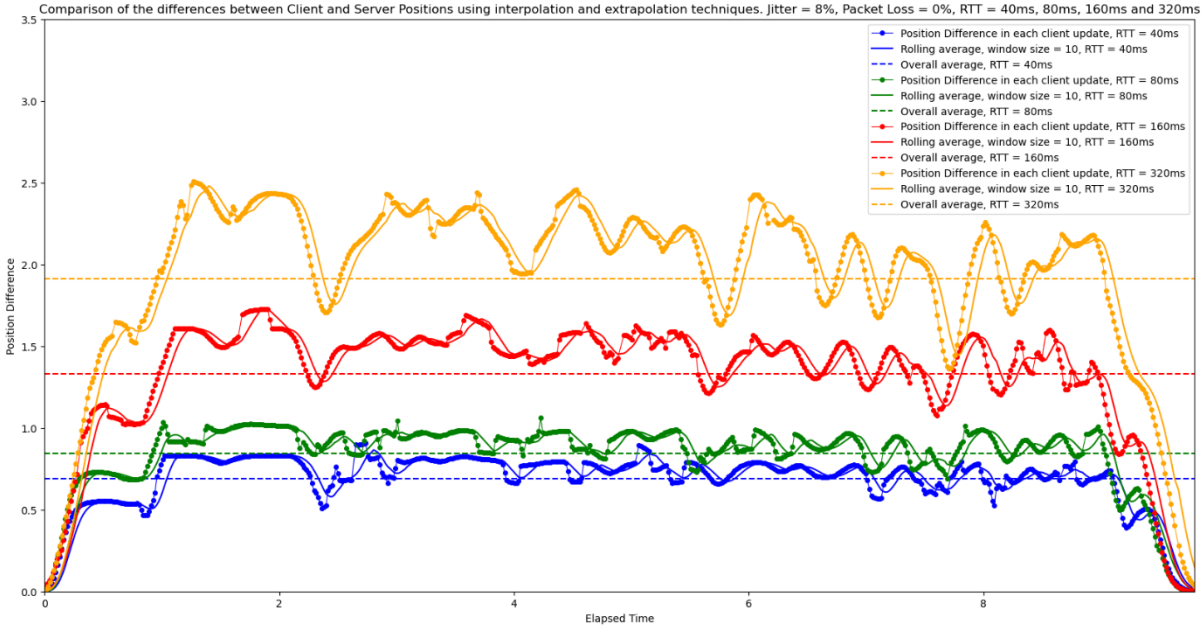


Ilustración 89 - Comparación de las diferencias entre las posiciones del cliente y el servidor usando los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 8% de jitter y 0% de pérdida de paquetes.

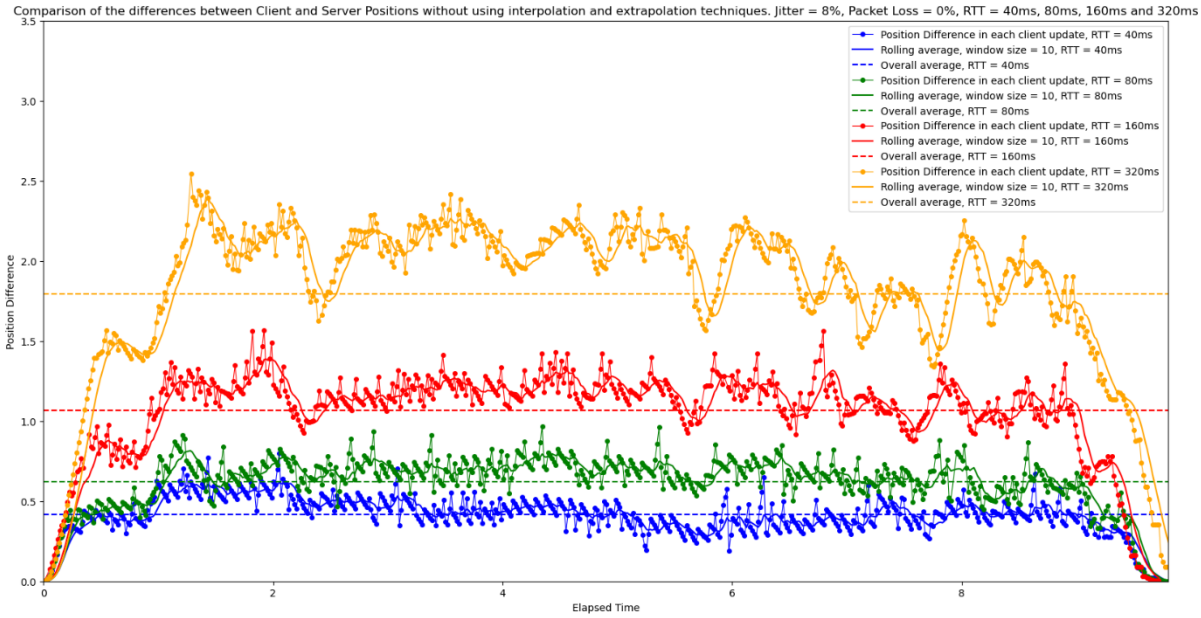


Ilustración 90 - Comparación de las diferencias entre las posiciones del cliente y el servidor sin usar los sistemas de interpolación y extrapolación de entidades en el cliente para los distintos valores de RTT (Azul = 40ms, verde = 80ms, rojo = 160ms y naranja = 320ms) usando un porcentaje de 8% de jitter y 0% de pérdida de paquetes.

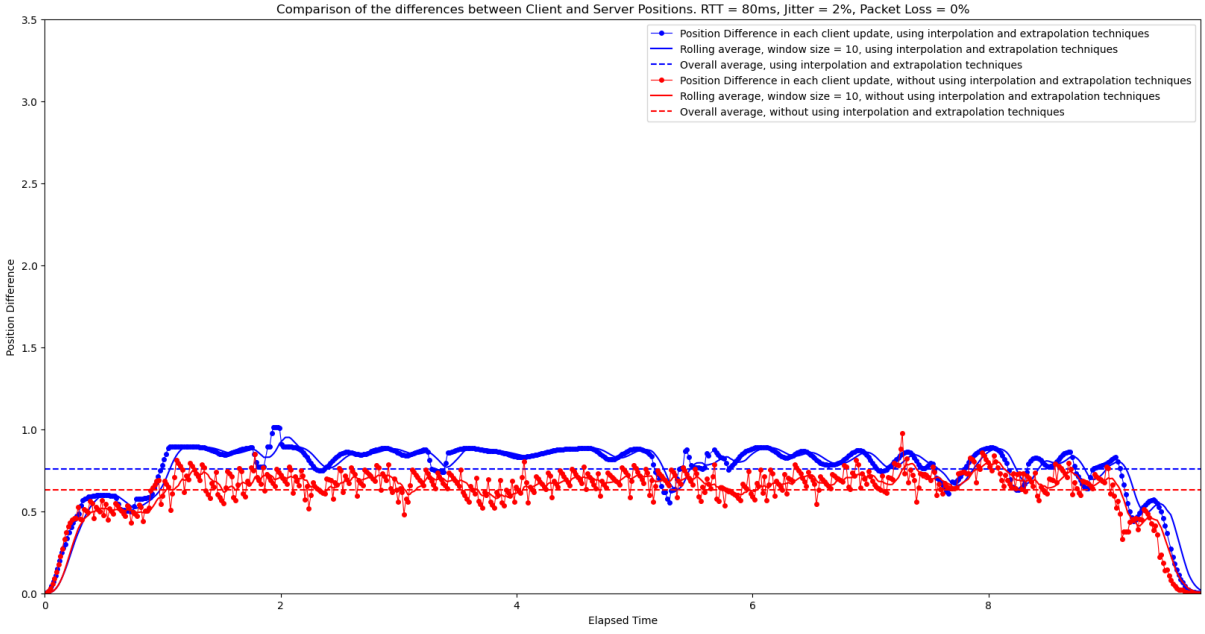


Ilustración 91 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 80ms usando un porcentaje de 2% de jitter y 0% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.

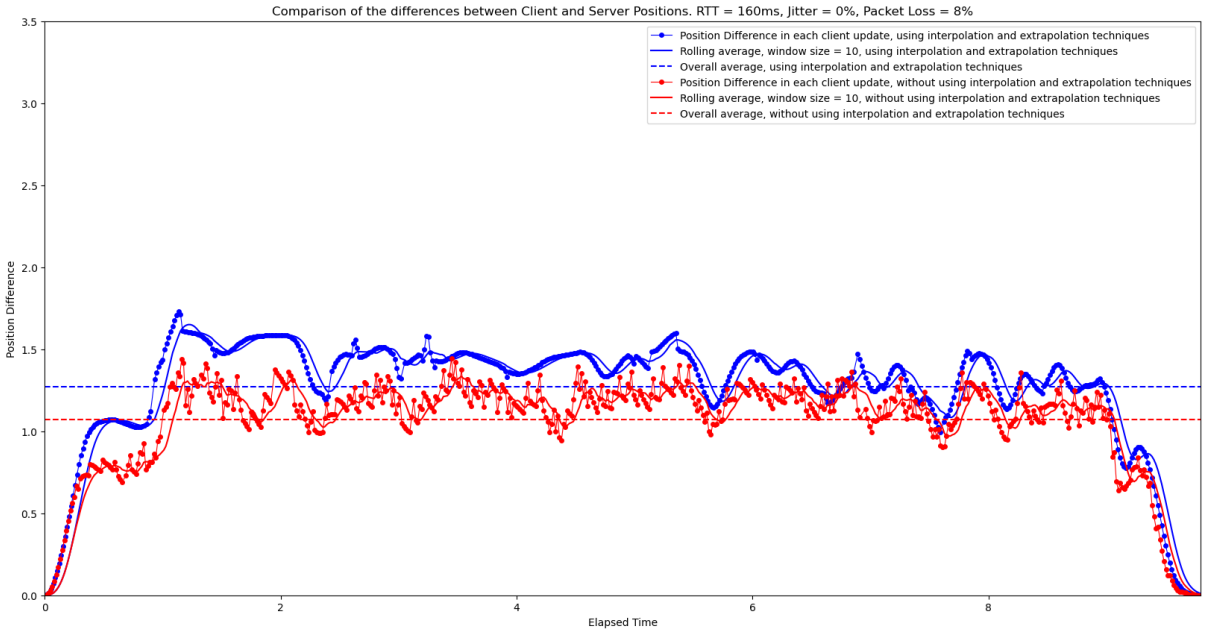


Ilustración 92 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 160ms usando un porcentaje de 0% de jitter y 8% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.

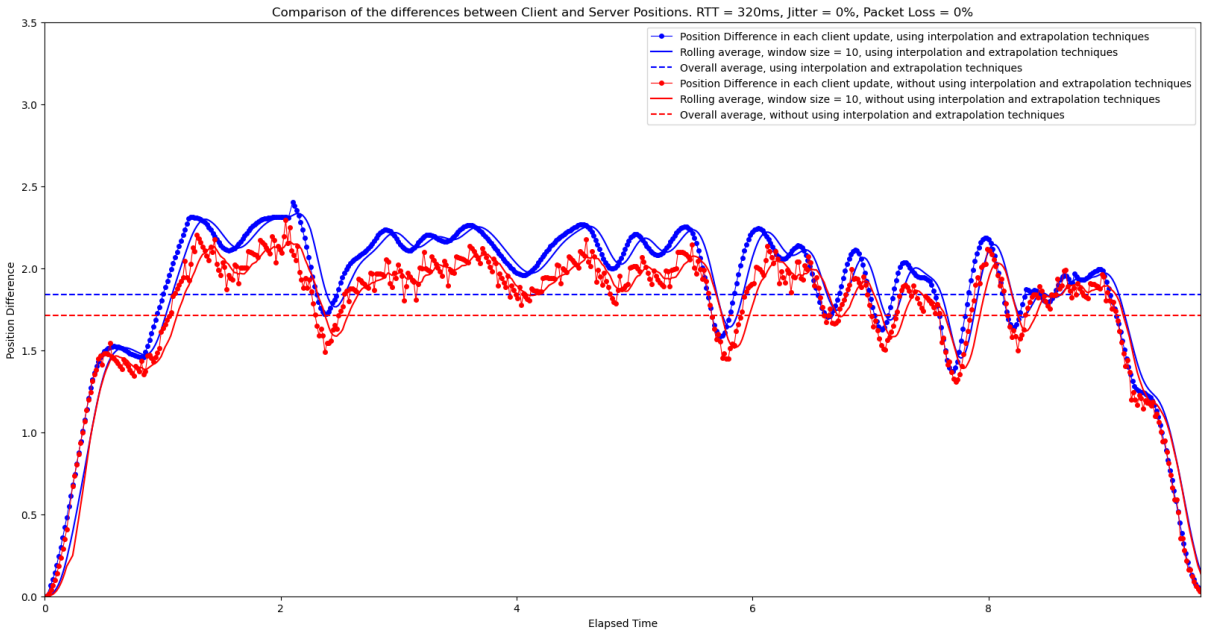


Ilustración 93 - Comparación de las diferencias entre las posiciones del cliente y el servidor para el caso en el que el RTT es igual a 320ms usando un porcentaje de 0% de jitter y 0% de pérdida de paquetes. Representadas en azul las diferencias de las posiciones usando los sistemas de interpolación y extrapolación de entidades en el cliente y en rojo sin ellos.