Universidad Rey Juan Carlos

Implementación de protocolos RUDP en el desarrollo de videojuegos multijugador

MEMORIA DEL TRABAJO DE FIN DE GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

Autora: Celtia Martín García

Tutora: María Teresa González de Lena Alonso

Curso: 2021-2022

Convocatoria: Julio 2022

Dedicado a
mi familia y amigos

Agradecimientos

Quería dar gracias a mi familia por siempre apoyarme y darme ánimos. También estoy muy agradecida a mis amigos y compañeros, y a su confianza en mí. Sobre todo, quiero agradecer a Javier Pérez Peláez, por ayudarme siendo mi “segundo jugador”.

También quiero dar gracias a los profesores de la Universidad Rey Juan Carlos por ayudarme en mi aprendizaje, sobre todo a mi tutora, María Teresa González de Lena Alonso, por haber visto potencial en mi idea de TFG y haberme apoyado hasta el final.

Resumen

Dentro del desarrollo de los videojuegos multijugador online, deben tenerse en cuenta muchos aspectos de la arquitectura de Internet, sobre todo para tomar decisiones de diseño acertadas y adaptadas a los requisitos. Una de ellas, de las más importantes a tomar, es qué tipo de protocolos van a utilizarse para la comunicación entre máquinas. Deben tomarse dos decisiones: qué protocolo va a encargarse del transporte de los paquetes, qué protocolos pueden incorporarse ya a nivel de aplicación para que el desarrollo sea más sencillo, y que el producto sea de mayor calidad. En cuanto a la primera decisión hay dos opciones: un protocolo de transporte fiable y muy completo (TCP), o un protocolo de transporte no fiable y simple (UDP). El primero asegura que los paquetes llegan al receptor y que lo hacen de forma ordenada, el segundo no, pero es mucho más veloz debido a su sencillez. Pero hay una tercera opción: los llamados protocolos RUDP(*Reliable UDP*), que utilizan UDP añadiendo una capa de fiabilidad a nivel de aplicación.

En este trabajo se va a implementar MUSE-RP, un protocolo RUDP orientado a videojuegos. Su diseño, fruto de una investigación sobre el uso de los protocolos RUDP en dicha área, se centró en una alta capacidad de configuración por parte del desarrollador, además de ofrecer dos canales de comunicación: uno fiable y otro parcialmente fiable.

Para poner a prueba este protocolo y su desempeño en un entorno real, se desarrolló un videojuego multijugador online sencillo. Este juego se ejecutó sobre varios protocolos: TCP, UDP sin protocolo intermedio, y tres protocolos RUDP: *GServer, Ruffles* y el propio MUSE-RP. De estas pruebas se extrajeron datos que se utilizarían en una comparativa del desempeño de estos protocolos.

Contenido

1. Introducción ... 1
1.1. Funcionamiento de las redes de computadores ... 1
1.2 Características de las redes en videojuegos ... 3
2. Estado del arte ... 7
2.1.1 Protocolos de transporte usados en videojuegos ... 7
2.1.2 Protocolos probados ... 11
3. Objetivos y tecnologías de desarrollo ... 13
3.1. Objetivos ... 13
3.2. Diseño del protocolo RUDP implementado: MUSE-RP ... 14
3.3. Tecnologías a utilizar ... 17
3.3.1 Lenguaje C# ... 17
3.3.2 Unity ... 17
3.3.3 Tecnologías de apoyo ... 18
3.4. Metodología ... 18
4. Implementación de MUSE-RP ... 21
4.1. Primera iteración ... 21
4.2. Segunda iteración ... 22
4.3. Tercera iteración ... 23
4.4. Iteración final ... 25
4.5. Pruebas y resultados ... 26
5.Diseño e implementación del juego de prueba ... 29
5.1. Diseño del gameplay del videojuego de prueba ... 29
5.2. Diseño de las comunicaciones del juego de prueba ... 33
6. Comparativa de protocolos ... 35
6.1. UDP ... 35
6.1.1. Comodidad al desarrollar ... 35
6.1.2. Experiencia empírica ... 36
6.1.3. Trazas *Wireshark* ... 37
6.2 Telepathy ... 38
6.2.1. Comodidad al desarrollar ... 38
6.2.2. Experiencia empírica ... 39
6.2.3. Trazas *Wireshark* ... 40
6.3 Ruffles ... 41
6.3.1. Comodidad al desarrollar ... 41
6.3.2. Experiencia empírica ... 42

6.3.3. Trazas *Wireshark* ... 42
6.4 GServer ... 43
6.4.1. Comodidad al desarrollar ... 44
6.4.2. Experiencia empírica ... 44
6.4.3. Trazas *Wireshark* ... 44
6.5 MUSE-RP ... 45
6.5.1. Comodidad al desarrollar ... 46
6.5.2. Experiencia empírica ... 47
6.5.3. Trazas *Wireshark* ... 47
6.6 Comparativa general y estadísticas ... 49
7. Conclusiones y trabajo futuro ... 51
8. Bibliografía ... 53
9. Anexo: Código y salidas por pantalla de las pruebas con MUSE-RP ... 55
9.1. Código de las pruebas ... 55
9.1.1. Código común ... 55
9.1.2. Prueba de chat sencillo ... 56
9.1.3. Prueba de envío masivo de mensajes ... 57
9.2. Salida por consola de las pruebas ... 58
9.2.1 Prueba de chat sencillo ... 58
9.2.2 Prueba de envío masivo de mensajes ... 59

Índice de ilustraciones

Ilustración 1: Representación de las capas de Internet, modelo IP/TCP y OSI 2
Ilustración 2: Árbol de decisión sobre qué protocolo de transporte escoger a la hora de diseñar un videojuego 8
Ilustración 3: Diagrama UML de las clases de MUSE-RP relacionadas con la gestión de puntos terminales 15
Ilustración 4: Diagrama UML de las clases de MUSE-RP relacionadas con la gestión de canales de comunicación 16
Ilustración 5: Resultados del análisis de código con SonarCloud para MUSE-RP 27
Ilustración 6: Paso por pantallas en el videojuego creado para las pruebas de los diferentes protocolos 31
Ilustración 7: Diagrama UML con las clases principales del juego de prueba, junto a sus atributos y métodos más destacables. 32
Ilustración 8: Puntuaciones erróneas producto de una partida al juego de prueba sobre UDP sin protocolo por encima 36
Ilustración 9: Muestra de paquetes capturados usando UDP en el juego de prueba 37
Ilustración 10: Gráfica de paquetes por segundo a lo largo de la comunicación con UDP 37
Ilustración 11: Muestra de paquetes TCP en la prueba del uso de Telepathy del videojuego con recepción de ACK duplicados y retrasmisión rápida 40
Ilustración 12: Muestra de paquetes TCP en la prueba del uso de Telapathy del videojuego 40
Ilustración 13: Gráfica de flujo de paquetes por segundo y errores de TCP a lo largo de la comunicación con TCP 40
Ilustración 14: Muestra de paquetes capturados en el transcurso de una partida usando Ruffles 42
Ilustración 15: Gráfica de paquetes por segundo en el transcurso de una partida usando Ruffles 43
Ilustración 16: Muestra de paquetes UDP usando GServer en el videojuego de prueba 44
Ilustración 17: Gráfica de flujo de paquetes por segundo a lo largo de la comunicación con GServer 45

Ilustración 18: Muestra de paquetes capturados usando MUSE-RP en el videojuego de prueba 47
Ilustración 19: Gráfica de paquetes por segundo a lo largo de la partida usando MUSE-RP 47
Ilustración 20: Salida por pantalla de la prueba de chat sencilla sobre MUSE-RP, con información sobre los mensajes de ping recibidos y el estado de la conexión 58
Ilustración 21: Salida por pantalla del servidor desconectándose de un cliente por timeout en MUSE-RP 58
Ilustración 22: Salida por pantalla de la creación y conexión de un cliente a un servidor MUSE-RP 59
Ilustración 23: Salida por pantalla de un caso de pérdida de mensajes por el canal parcialmente fiable en MUSE-RP 59

Índice de algoritmos

Algoritmo 1:Clase MessageObjectComparer, que implementa la interfaz IComparer para los objetos MessageObject 24
Algoritmo 2:Bucle de escucha de mensajes implementado para el cliente Ruffles 42
Algoritmo 3:Código de creación de un cliente de las primeras pruebas de MUSE-RP . 55
Algoritmo 4: Código de creación de un servidor de las primeras pruebas de MUSE-RP 55
Algoritmo 5: Manejador de recepción de mensajes de chat por parte del servidor........ 56
Algoritmo 6: Manejador de recepción de mensajes de chat por parte del cliente 56
Algoritmo 7: Hilo de envío de mensajes escritos por parte del cliente.......................... 56
Algoritmo 8: Hilo de envío de mensajes escritos por parte del servidor 56
Algoritmo 9: Manejador de recepción de mensajes masivos por parte del cliente 57
Algoritmo 10: Hilo de envío masivo de mensajes por parte del servidor...................... 57

Índice de tablas

Tabla 1: Estadísticas de los protocolos en el desempeño de una partida online en el videojuego de prueba 49

1. Introducción

Las redes y las aplicaciones que usan Internet están muy presentes en la vida cotidiana. Es un tema muy interesante, a la vez que desafiante, sobre todo en la industria informática. Los videojuegos son parte de este mundo, por tanto conocer las características generales del mismo es esencial para poder abordar temas más específicos relacionados con el desarrollo de videojuegos online.

1.1. Funcionamiento de las redes de computadores

Para un resumen sobre el funcionamiento de las redes, se va a emplear del libro “Computer Networking. *A Top-Down*.”[1]. Hoy en día se vive rodeado de dispositivos con acceso a diferentes tipos de redes, siendo la más importante y conocida Internet. Internet la conforman millones de puntos terminales (máquinas con acceso a ella), junto a los enlaces que las conectan entre ellas. Una conexión punto a punto (entre dos de estas máquinas) es posible gracias a dichos enlaces y a conmutadores de paquetes (como pueden ser los *routers*).

Para una descripción más estructurada, se decidió dividir el funcionamiento de las redes en capas abstractas. Estas capas son representaciones del paso de información por los diferentes niveles de la red: nivel físico, nivel de enlace, nivel de red, nivel de transporte y nivel de aplicación. Este modelo de capas puede llamarse también pila de protocolos, ya que cada capa usará un protocolo diferente. Cada paquete de estos protocolos encapsulará siempre información sobre las capas superiores, por ello se dice que las capas superiores se benefician de los servicios de las inferiores.

Los protocolos más importantes (y cuyo conocimiento será necesario para el desarrollo de este trabajo) son: el protocolo IP (*Internet Protocol*) en la capa de enlace, que suministra direcciones a todos los dispositivos de la red; y el protocolo TCP (*Trasmission Control Protocol)* en la capa de transporte, un protocolo muy complejo y completo que se encarga de transportar los paquetes de un punto terminal a otro. Por la importancia de estos protocolos, a veces al modelo de 5 capas se le llama modelo TCP/IP.

Pero TCP no es el único protocolo con el que cuenta la capa de transporte. También existe UDP*(User Datagram Protocol*), un protocolo simple y ligero, que no cuenta con tantas funcionalidades como TCP, pero que lo compensa con su baja latencia. Sobre estos

dos protocolos se hablarán más en el apartado 2.1.1 Protocolos , dada su importancia para el desarrollo de este documento.

La capa de aplicación cuenta también con otros protocolos conocidos, tales como pueden ser HTTP para las peticiones web, o FTP para la descarga y subida de archivos. También cabe destacar que, para la comunicación entre la capa de transporte y la de aplicación, existe una interfaz muy importante para el desarrollo de aplicaciones que hacen uso de la red, llamada *socket.*

Por último, aparte de las capas descritas, también hay modelos que incorporan capas entre la capa de transporte y de aplicación, como pueden ser la capa de presentación o de sesión. Puede verse estos dos modelos capas en la Ilustración 1.

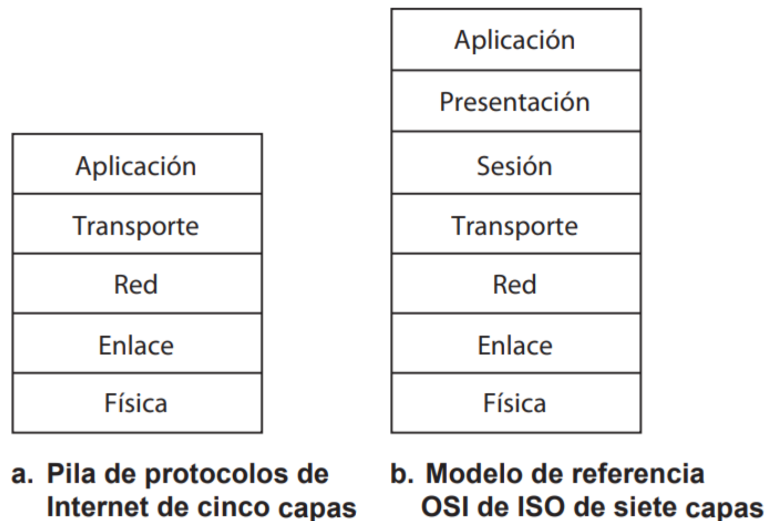

Ilustración 1: Representación de las capas de Internet, modelo IP/TCP y OSI

1.2 Características de las redes en videojuegos

Con los conocimientos básicos sobre el funcionamiento de Internet, ya se puede hablar de cómo su arquitectura influye en los videojuegos online.

Los videojuegos son parte de la capa de aplicación, ya que estos se nutrirán de los servicios de todas las capas de la red para poder comunicarse. Por esto mismo, a los desarrolladores de videojuegos debería importarles, sobre todo, dos de las capas que conforman Internet: la capa de aplicación y la capa de transporte. Esto también es debido a que son las únicas a las que pueden llegar a tener acceso.

Hay dos grandes cuestiones que deben resolverse para entender mejor cómo usan Internet los videojuegos:

- **¿Qué tipo de mensajes se envían las máquinas participes en un juego online?** Esencialmente, eventos e información sobre el entorno del juego. Estos mensajes pueden llevar información sobre, por ejemplo, cambios en el mundo del juego, que un jugador ha sido derrotado o que al usuario le han hecho daño.
- **¿Quién es el encargado de mandar dichos mensajes?** Eso depende de la arquitectura específica del videojuego. Es muy común la arquitectura Cliente-Servidor con un Servidor Autoritativo. Este tipo de servidor se usa para evitar trampas[2],y consiste en que es el servidor quién se encarga de toda la lógica del videojuego (como por ejemplo las físicas, los eventos o las colisiones) en base a las entradas que reciba de los jugadores (como elementos de la interfaz activados o botones pulsados). Los usuarios necesitan más ancho de banda para los mensajes de llegada que para los de salida, ya que el servidor estará mandando continuamente información al jugador, mientras que este se limitará a mandar las acciones que él mismo esté realizando. [3]

La arquitectura *Peer to Peer* (punto a punto) también es usada en el entorno de los videojuegos. Es una arquitectura muy prometedora y no es tan costosa como la arquitectura Cliente-Servidor, pero aún debe evolucionar más para poder ser una opción más viable a la hora de desarrollar videojuegos [4].

Para el desarrollo de un modo multijugador en un videojuego, también hay que considerar el propio diseño del videojuego, ya que hay que tener en cuenta que no todos los juegos requieren de los mismos recursos de red. Póngase el ejemplo de dos juegos online: un juego de ajedrez y un juego de lucha (como, por ejemplo, *Mortal Kombat*).

En cuanto al primero:

- Es un juego lento.
- No requiere de un tráfico muy concurrido entre los puntos terminales que participen en una partida, ya que la única información que necesitan enviar son los cambios de posición de las piezas.
- No se puede permitir perder información en la comunicación entre dos jugadores, ya que causaría un error fatal en el transcurso de la partida.

Sin embargo, en el caso del juego de lucha:

- Es un juego muy dinámico, que bombardea al usuario con información gráfica.
- Debido a este bombardeo continúo de información, el tráfico entre las máquinas de los jugadores participantes será bastante más denso que en el caso anterior.
- Mientras que algunos mensajes son imprescindibles (como el mensaje de *K.O.* o de que se ha producido un golpe), otros pueden llegar a poder perderse sin mayor problema, siempre y cuando las pérdidas no sean demasiadas (como, por ejemplo, los mensajes de posición del contrincante).

El desarrollo del apartado de comunicaciones de estos dos juegos deberían tener una perspectiva totalmente diferente. Se deberá pensar en qué se debe priorizar en cada caso y que tecnologías se adaptan más a su diseño.

Centrémonos un poco más en el segundo tipo de juego, ya que, a primera vista, es el que tiene los requisitos más exigentes y difíciles de conseguir. La latencia afecta enormemente a la experiencia del jugador, ya que, al estar sumergido en un mundo interactivo, cualquier deficiencia en el *feedback* recibido dificultará la inmersión del usuario[5]. En un juego dinámico *peer to peer*, por ejemplo, el jugador empieza a percibir

deficiencias a partir de los 100 ms de retraso de paquetes y de un porcentaje de pérdidas del 10%[6].

En el apartado 2.1.1 Protocolos se volverá a hablar de estos dos ejemplos, ya que dependiendo de su diseño, los protocolos que se decidan usar para cada uno de los juegos puede variar.

Aparte de esta clasificación, en función del dinamismo del videojuego, algunos estudios también clasifican los juegos multijugador como *Transient* o *Persistent,* dependiendo de si necesitan mantener un estado persistente en un mundo constante (como podría ser el ejemplo de un MMORPG (juego de rol multijugador online masivo), o si no lo necesitan por tener sesiones de juego cortas y dinámicas (como podría ser, por ejemplo, el juego de lucha descrito en los párrafos anteriores)[7]. Esto también es importante, porque no se puede permitir que un juego persistente vaya acumulando errores en largas sesiones de juego[8]. Por tanto, este tipo de clasificación también debe tenerse en cuenta a la hora de tomar decisiones sobre el apartado de comunicaciones de un videojuego multijugador.

Juntando todo este tipo de cuestiones sobre las redes en el entorno de los videojuegos, puede llegarse a una conclusión clara: el tráfico de los juegos online no se asemeja a ningún otro y es muy diverso y variable. Es decir: no hay un estándar. El elemento clave para esta diferenciación con los demás tipos de tráfico es la interacción del jugador con el entorno del juego.

Pongamos el caso del tráfico web. Un usuario puede interactuar con la interfaz de dicha web, pero esta interacción es a veces lenta y torpe. De hecho, normalmente se debe esperar entre interacciones a que se cargue el contenido de la web. Un videojuego no puede permitirse tal lentitud, ya que el usuario está inmerso dentro de él, y cualquier tipo de retraso le sacaría de la experiencia de juego[5]. Esto también añade nuevos retos en el desarrollo en red. Véanse algunos de los más importantes:

- **Posibles fallos de concurrencia.** En los videojuegos un fallo de concurrencia puede ser crítico y resultar en una ejecución errónea. Para poner en contexto, un fallo de concurrencia es un fallo derivado de la ejecución de varios procesos en paralelo que se comunican entre sí. Póngase un ejemplo: si un jugador dispara a otro en un momento T, y el otro jugador decide usar algún tipo de objeto para curar su salud en el mismo momento del impacto de dicho disparo, ¿qué evento debería ejecutarse primero? En este caso, el orden de los factores

altera el producto. Si la curación ocurre antes, el jugador disparado podría sobrevivir. Sin embargo, si es el disparo el que ocurre primero, el jugador podría perder la partida. Podría ocurrir una tercera opción: el disparo ocurre primero, pero el jugador se cura igualmente, pudiendo causar un comportamiento no deseado si el evento de muerte ha sido disparado pero el jugador puede seguir jugando.

- **Posibilidad de jugadores que hagan trampas.** Como ya se ha comentado, esto puede intentar solucionarse con un servidor autoritativo.
- **Problemas a la hora de refrescar información** (sobre todo gráfica), también conocido como *lag*. Los videojuegos tienen, sobre todo, estímulos visuales, y como ya se sabe, el cerebro humano necesita de unas 24 imágenes por segundo para crear una sensación de movimiento fluido. Mandar la posición de un jugador cada 41 ms podría llegar a saturar la red, además, de que si se hace por un canal no fiable, podrían perderse paquetes. Es por ello que muchos juegos aplican mecanismos para interpolar dicho movimiento. Hablemos de dos de estos mecanismos : *Snapshot* y compresión [9], que se centran en comprimir la información y crear "instantáneas" del movimiento; y *Dead Reckoning* [10], que consiste en que cada ejecución del juego interpola y simula el movimiento de los demás jugadores, utilizando la información recibida como corrección a dicha interpolación. Estas herramientas pueden ser muy interesantes de estudiar, sin embargo en este trabajo no se tendrán en cuenta más allá de mencionarlas y tenerlas presentes, ya que se va a trabajar a un nivel más bajo de abstracción, cerca de la capa de transporte anteriormente mencionada.

Ahora que ya se ha profundizado y ya se tienen más conocimientos sobre el entorno de estudio, es el momento de hacer una investigación más técnica y descubrir qué tecnologías se están utilizando en los videojuegos.

2. Estado del arte

Teniendo en mente todas las particularidades de los videojuegos explicadas en el apartado anterior, se estudiará con más profundidad la situación actual del desarrollo de videojuegos online, haciendo especial hincapié en qué protocolos y tecnologías son las posibles soluciones a los problemas que plantean este tipo de aplicaciones.

2.1.1 Protocolos de transporte usados en videojuegos

Véase un pequeño resumen de en qué consisten los dos protocolos disponibles en la capa de transporte:

- **TCP** es el protocolo más completo que hay en la capa de transporte. Es fiable (se garantiza que los paquetes llegan a su destino y que lo hacen ordenadamente), cuenta con mecanismos muy interesantes, tales como el control de congestión de la red y control de flujo. Pero esto puede ser un arma de doble filo, porque este tipo de características no pueden desactivarse, y pueden causar una alta latencia, sobre todo por el llamado bloqueo HOL(*Head of Line Blocking*)[11]. Este tipo de bloqueos son causado porque, tal y como TCP funciona, si un paquete se pierde todos los demás recibidos no pueden pasar a capa de aplicación, creando una congestión en el tráfico que no terminará hasta que el emisor decida reenviar dicho paquete. Pero, ¿por qué sabe TCP que un paquete ha llegado a su destino?, pues porque el receptor de los paquetes debe enviar un mensaje de reconocimiento (también llamado ACK), que confirma la llegada de la información recibida.[1]
- **UDP**, por otro lado, es muy sencillo, que no cuenta con funcionalidades complejas. Hace su mejor esfuerzo por entregar los paquetes enviados, pero no garantiza orden ni entrega. Es decir, no es fiable. Esto lo compensa con su baja latencia, lo cual lo hace muy atractivo para aquellos que deseen primar la velocidad ante la fiabilidad.[1]

En el trabajo “Estudio y diseño de protocolos RUDP orientados a videojuegos.”[12] ya se habla de la disputa entre TCP y UDP en el mundo de los videojuegos online, pero aquí se debe reincidir en ella.

Con lo visto en el apartado 1.2 Características de las redes en videojuegos puede adivinarse que algunos desarrolladores preferirán usar TCP, mientras tantos otros optarán por UDP. No hay una respuesta correcta a la pregunta de qué protocolo es mejor, ya que esto depende de muchos factores del propio videojuego que se quiera desarrollar. Una manera simple de resumir esta decisión puede verse en la Ilustración 2.

Recalcar que no es una decisión tan simple y que deben tenerse en cuenta otros factores, pero este diagrama puede ayudar en la decisión final.

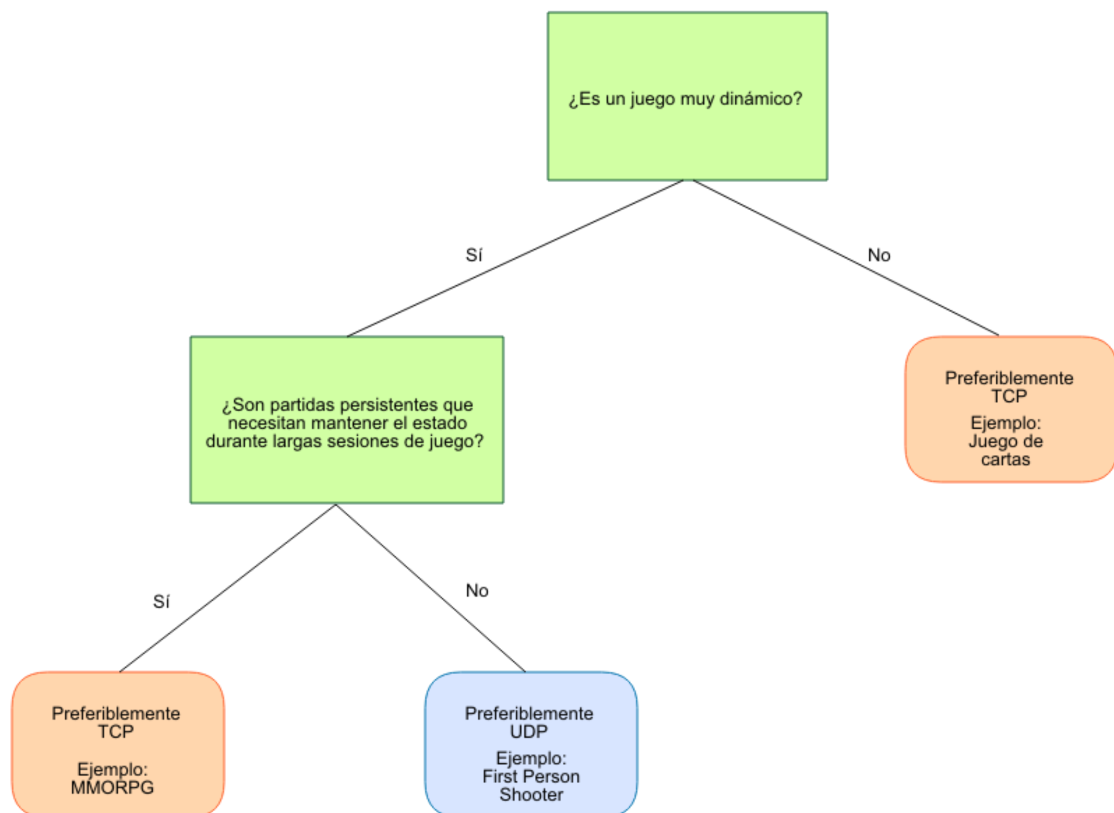

Ilustración 2: Árbol de decisión sobre qué protocolo de transporte escoger a la hora de diseñar un videojuego

¿Qué usan los juegos más exitosos en la actualidad? Véanse unos cuantos:

- *Minecraft*, un *sandbox* que consiste en sobrevivir y construir en un mundo formado por cubos, tiene posibilidad de crear y/o unirse a un servidor, el cual usará TCP para comunicarse. Este es un ejemplo de que el diagrama de la Ilustración 2 no es determinante. *Minecraft*, de hecho, aplica mecanismos para

hacer que TCP tenga mejor rendimiento. Uno de ellos, por ejemplo, es el uso de paquetes de poco tamaño [8][12].

- *World of Warcraft*, uno de los MMORPG (Juegos de rol masivos online) más famosos, usa TCP. Esto sí concuerda con el árbol de decisión implementado, ya que es un juego persistente que no puede permitirse acarrear errores, y que prima que la entrada del usuario sea válida a que la comunicación sea veloz[8].
- *Stardew Valley* es un juego de gestionar tu propia granja, a la cual puedes invitar a tus amigos, y para las comunicaciones usa UDP[12]. Esto parece razonable, ya que es un juego con bastante dinamismo, donde los jugadores pueden verse ejecutando acciones tales como minar, pelear con enemigos o plantar semillas.

Ahora bien, hay una alternativa a TCP y UDP : los protocolos RUDP (*Reliable UDP*). Estos protocolos seleccionan lo mejor de TCP y se lo incorporan a UDP en capa de aplicación[13], creando una “subcapa fiable” entre la aplicación y el transporte. ¿Por qué no crear directamente un protocolo de transporte? Porque la capa de transporte está bastante restringida a los desarrolladores, e implementar un protocolo de transporte no es trivial. De hecho, algunos programadores comenzaron con la idea de crear un protocolo de transporte y se toparon barreras que les impidieron continuar, por tanto decidieron pasar la funcionalidad a capa de aplicación y usar UDP, creando un RUDP. Es el caso, por ejemplo, de GTP(*Game Transport Protocol*)[14].

Algunos proclaman, de hecho, que este protocolo es la solución perfecta para implementar videojuegos[15][16], y se sabe que actualmente se están utilizando profesionalmente [15]. Un ejemplo podría ser, presuntamente, el propio *Stardew Valley*. Antes se mencionó que utiliza UDP, pero, al no ser de código abierto, no se puede saber con certeza si no se está usando un protocolo fiable por encima, pero un estudio más concienzudo de su tráfico [12] revela que la posibilidad de que se esté usando un protocolo RUDP es muy alta, ya que algunos paquetes parecen estar actuando como ACKs.

Los argumentos a favor de los algoritmos RUDP se centran en su alta adaptabilidad a todo tipo de juego, y a qué es el híbrido perfecto entre fiabilidad (TCP) y velocidad (UDP). Sin embargo, no hay un solo protocolo RUDP. No hay un estándar, por tanto se pueden encontrar protocolos de todo tipo.

Este abanico de posibilidades ofrece al desarrollador un amplio catálogo, del que podrá elegir el protocolo que más se adapte a las necesidades del juego que quiera desarrollar. Y en caso de no encontrar el adecuado, siempre puede desarrollarse uno que se ajuste a las características deseadas, aunque esto añadiría bastante complejidad al proceso de desarrollo. Ya en "Estudio y diseño de protocolos RUDP orientados a videojuegos."[12] se llegó a la conclusión de que los protocolos RUDP son una herramienta muy valiosa y con mucho futuro en el campo de los videojuegos.

Una última cuestión que hay que abordar es cómo se puede añadir fiabilidad a UDP. Para ello puede implementarse, por ejemplo, un algoritmo de ventana deslizante. No se va a profundizar mucho en ello en este trabajo, pero por hacer un breve resumen, tiene que ver con los paquetes ACK mencionados anteriormente. Tanto el emisor tiene una "ventana de paquetes" de cierto tamaño, con paquetes enviados que el emisor aún no ha reconocido. El emisor no puede enviar paquetes si la ventana está llena, y esta se irá vaciando o "deslizando" si el paquete más antiguo ha sido reconocido. Hay dos maneras de abordar este tipo de algoritmos: de manera selectiva, por lo que el receptor reconocerá cada paquete individualmente; o acumulativa, donde el receptor solo reconocerá los paquetes hasta que tenga el paquete más antiguo no recibido. Para esta segunda opción el receptor puede contar con un buffer de llegada, donde guardará los paquetes que han llegado desordenados, pero que no serán reconocidos hasta que se rellenen los huecos. Cuando en el receptor reconoce los paquetes, los pasa a capa de aplicación. Por último, cabe destacar que en el caso del reconocimiento acumulativo, cuando se recibe el reconocimiento de un paquete, puede decirse con seguridad que todos los anteriores a él también han llegado. TCP usa reconocimiento acumulativo.

En el siguiente apartado se hablará en más profundidad de los protocolos de código abierto que han podido probarse en un entorno real de desarrollo sencillo, mientras que en el capítulo 4. Implementación de MUSE-RP se hablará de la implementación del protocolo que se diseñó en "Estudio y diseño de protocolos RUDP orientados a videojuegos."[12], llamado MUSE-RP(*Multiplayer UDP Service Extension-Reliable Protocol*), que también será puesto a prueba en el mismo entorno que los demás.

2.1.2 Protocolos probados

En esta sección se va a hablar un poco de los protocolos que se han estudiado y probado en este trabajo. Se han probado un total de 3 protocolos RUDP: *GServer*[17], *Ruffles* [18], y el propio MUSE-RP [12]. Además, se ha querido comparar con UDP sin protocolo por encima, y TCP. Para la inclusión de TCP se ha usado un protocolo llamado *Telepathy[*19], específico para videojuegos, ya que hizo más sencillo el desarrollo de las pruebas. Véase un resumen de lo que ofrecen estos protocolos:

- ***GServer*** es un protocolo RUDP, flexible, sencillo y específico para videojuegos. Ofrece varios modos de envío, pudiendo combinar fiabilidad en la entrega y en el orden al gusto del desarrollador. Marca los paquetes con su tipo específico y su prioridad en cuanto a su procesado, y usa ACKs selectivos. Está escrito en C#.[17]
- ***Ruffles*****,** por otro lado, es un protocolo RUDP muy completo con muchas características interesantes, también implementado sobre C#. Entre dichas características se cuenta con: posibilidad de uso de direcciones Ipv6, estadísticas de conexión, fragmentación, manejo de conexión, seguridad para usar en hilos, buen rendimiento, y seguimiento del ancho de banda utilizado. También utiliza C# y es seguro utilizarlo en *Unity*.[18]
- ***Telepathy,*** sin embargo, es un protocolo sobre TCP. Facilita el uso de este protocolo de transporte en un videojuego. En el README que ofrece su repositorio queda claro que su especialidad son los MMO, es decir, los juegos multijugador masivo online. También está escrito en C#, y está diseñado siguiendo el principio KISS(*Keep It Simple Stupid*). De sus características más interesantes (alto rendimiento, simpleza, multihilo), la que fue determinante para su uso en este trabajo fue que incorpora fragmentación en mensajes. ¿Por qué es esto importante? Porque TCP trabaja con "flujos de datos" o *streams*, lo que añade una gran dificultad en su uso en videojuegos, ya que al pasar un paquete de capa de transporte a capa de aplicación no se puede saber si lo que ha llegado ha sido un mensaje o varios fusionados. [19]

- De **MUSE-RP** se hablará en el apartado 3.2. Diseño del protocolo RUDP, pero véase un resumen de algunas de sus características: cuenta con dos canales de comunicación (uno fiable y otro parcialmente fiable), usa ACKs acumulativos y cuenta con mecanismos orientados a conexión. También está escrito en C#, y puede verse un diseño más elaborado en “Estudio y diseño de protocolos RUDP orientados a videojuegos.”[12].

3. Objetivos y tecnologías de desarrollo

En este apartado se establecerán los objetivos de este trabajo, se hablará un poco del diseño de MUSE-RP y se explorarán las tecnologías y la metodología que se van a aplicar en este trabajo.

3.1. Objetivos

El objetivo principal de este trabajo es implementar un protocolo RUDP previamente diseñado (MUSE-RP[12]) y probarlo, junto a otros protocolos RUDP, en un entorno real.

Aparte, se desean conseguir los siguientes objetivos secundarios:

- Crear un minijuego multijugador online sencillo, donde varios usuarios puedan conectarse y jugar una partida. Teniendo en cuenta el árbol de decisión del apartado 2.1.1 Protocolos de transporte usados en videojuegos, este juego será, preferiblemente, dinámico, dado que se trata de probar protocolos RUDP es un entorno más favorable.
- Implementar un protocolo RUDP siguiendo el diseño de MUSE-RP[12] , incluyendo el correcto funcionamiento de todas sus funcionalidades descritas.
- Hacer una comparativa de trazas, comodidad de uso y funcionalidades incorporadas entre MUSE-RP y otros protocolos RUDP de código abierto.
- Hacer una segunda comparativa entre dichos protocolos RUDP, TCP y UDP sin protocolo asociado.

3.2. Diseño del protocolo RUDP implementado: MUSE-RP

El diseño completo de MUSE-RP se explica con detalle en "Estudio y diseño de protocolos RUDP orientados a videojuegos"[12], pero se proporcionará un resumen de sus funcionalidades junto a los diagramas de clases UML (Ilustración 3,Ilustración 4) para entrar en contexto.

Características fundamentales de MUSE-RP:

- Protocolo fiable y/o parcialmente fiable sobre UDP.
- Uso de dos canales de comunicación, uno fiable y otro parcialmente fiable. Este canal parcialmente fiable podrá tolerar pérdidas hasta un porcentaje estipulado por el desarrollador.
- Implementación de dos algoritmos de ventana deslizante (uno fiable y otro parcial), inspirados en el algoritmo fiable de TCP[1].
- Mecanismos orientados a conexión (inicio de conexión por medio de un *handshake* por cada canal, final de conexión, y mecanismos de *timeout* en caso de inactividad)
- Otras funcionalidades inspiradas en TCP, tales como: retrasmisión rápida (retrasmisión del paquete al recibir 3 ACKs duplicados),o cálculo de RTT (gracias a la implementación de un mecanismo propio de ping) y modificación del temporizador de retrasmisión según el mismo.
- Uso de manejadores por cada tipo de paquete, que aumentan la comodidad del desarrollador a la hora de implementar la funcionalidad de los mensajes.
- Configurable y orientado a una arquitectura Cliente-Servidor.

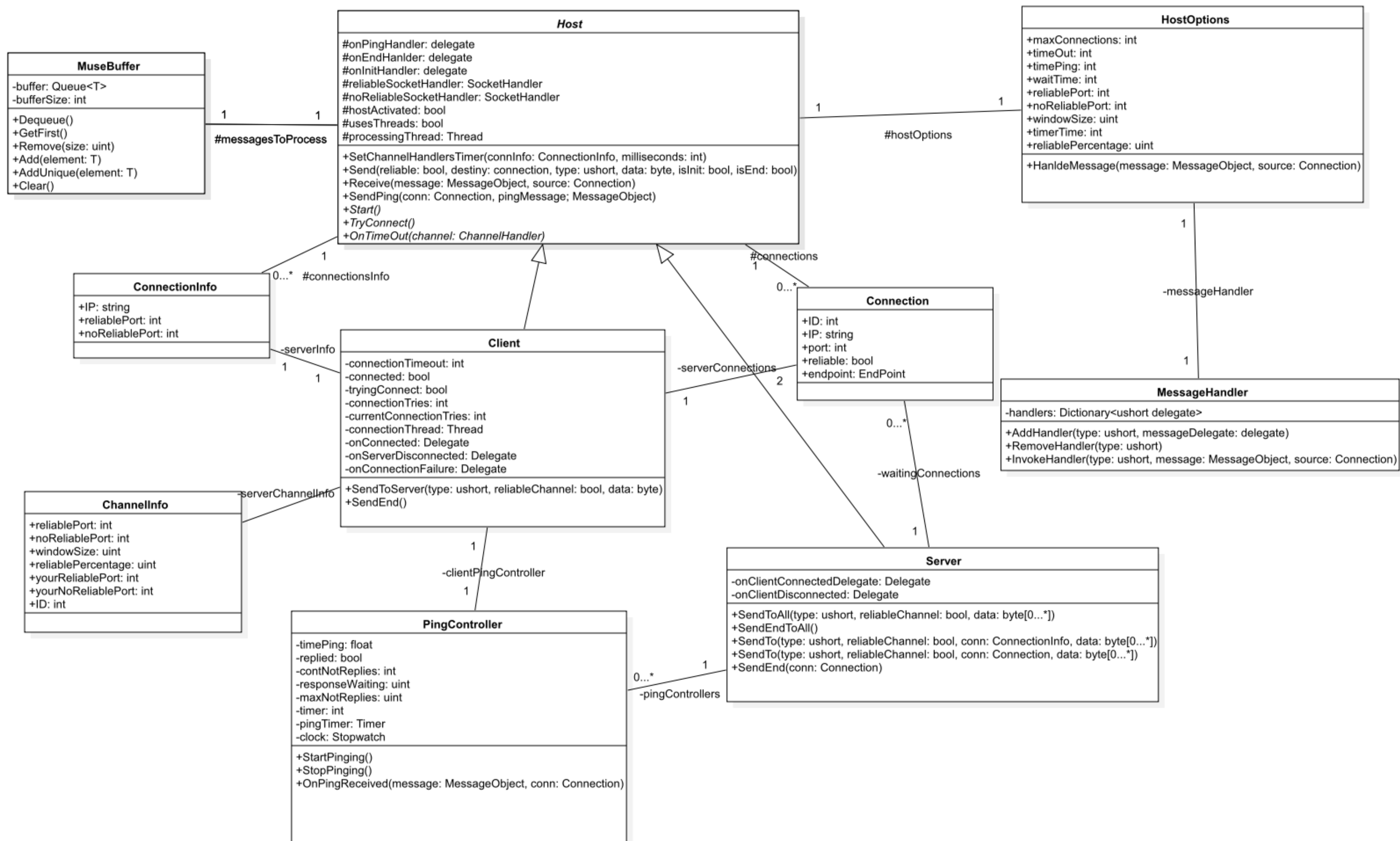

Ilustración 3: Diagrama UML de las clases de MUSE-RP relacionadas con la gestión de puntos terminales

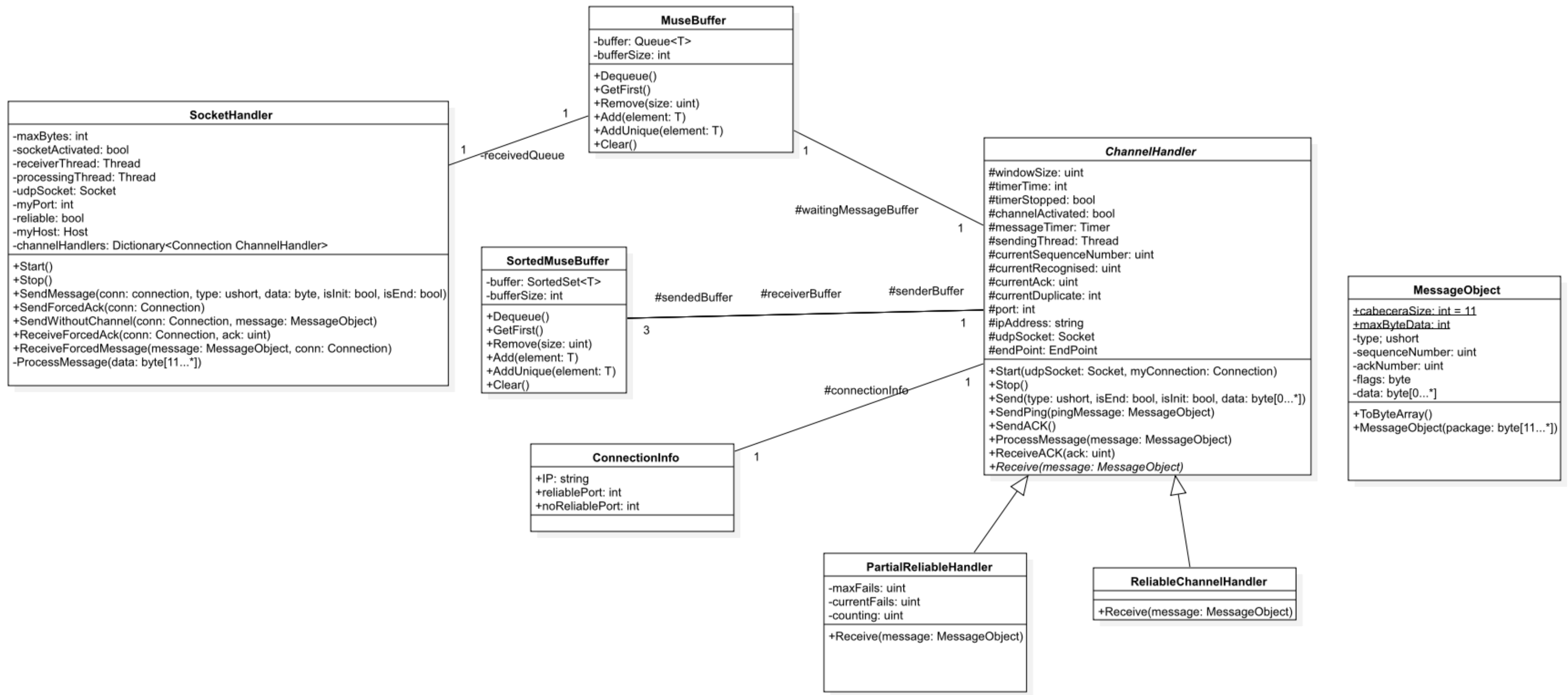

Ilustración 4: Diagrama UML de las clases de MUSE-RP relacionadas con la gestión de canales de comunicación

3.3. Tecnologías a utilizar

A continuación va a hablarse de las diferentes tecnologías que serán útiles a lo largo del desarrollo de este trabajo.

3.3.1 Lenguaje C#

C# es un lenguaje de alto nivel, orientado a objetos y a componentes, con seguridad de tipos y con un fuerte control de versiones [20]. Podría decirse que es un híbrido entre C, C++ y Java, y es un lenguaje muy común en el ámbito del desarrollo de videojuegos[21] por su eficiencia y escalabilidad. Cuenta con muchas herramientas muy potentes a la hora de desarrollar aplicaciones online, como por ejemplo: clases orientadas al uso de *sockets*, al manejo de hilos, y al control de problemas de concurrencia.

Además de estas características, las cuales ya le hacen muy atractivo para su uso en este trabajo, es el lenguaje que utilizan muchos de los protocolos RUDP (ver 2.1.2 Protocolos probados), y el que utiliza Unity, que será la herramienta que se usará para el desarrollo del juego de prueba.

Se utilizó el *framework* de .NET, uno de los más utilizados, junto al IDE *Visual Studio*.

3.3.2 Unity

Unity es un motor multiplataforma para desarrollar videojuegos muy usado actualmente en la industria, sobre todo en el mundo del desarrollo de juegos indies y de los nuevos desarrolladores[22].

Es una herramienta potente y fácil de usar, además de que ofrece una versión gratuita para uso personal. Puede usarse tanto para entornos 2D como 3D, y cuenta con un *Asset Store* con un gran catálogo de herramientas y *assets*. Entre estas herramientas pueden encontrarse APIs de *networking* tales como *Mirror*[23] o *Photon*[24]. Estas APIs son de alto nivel, por encima de la capa de transporte e incluso de la capa que crean los algoritmos RUDP. Aportan muchas funcionalidades útiles para los desarrolladores, tales como clases específicas para crear componentes cuyos datos deben compartirse entre los participantes de un juego, herramientas para la creación de partidas, o la elección sobre qué protocolos usar. De hecho, *Mirror* ofrece soporte para poder utilizar, por ejemplo,

Telepathy (ver 2.1.2 Protocolos probados) o *kcp*, que es un protocolo RUDP [12][25]. En este trabajo se ha intentado no utilizar estas herramientas, ya que se quería realizar la comparativa de los protocolos a más bajo nivel posible, sin una capa de abstracción adicional.

Por último, y como ya se ha dicho en el apartado anterior, utiliza C#, lo que quiere decir que el desarrollo de MUSE-RP no debería encontrarse con muchos problemas de compatibilidad.

3.3.3 Tecnologías de apoyo

Otros softwares que se han utilizado en el desarrollo de este trabajo:

- *Wireshark*[26], un programa de captura de paquetes, que no solo ha sido utilizado para la creación de trazas en las diversas pruebas de cada protocolo, si no que ha sido útil para el desarrollo de la implementación de MUSE-RP. Esto es debido a que el desarrollo de aplicaciones que usan Internet es tedioso, y a veces no se sabe si los paquetes se están mandando o recibiendo correctamente, problema que solucionan los softwares de captura de paquetes.
- *Hamachi*[27], un software que crea una red privada entre varios puntos terminales. Este programa ha sido útil a la hora de hacer pruebas sin que el servidor tenga que abrir los puertos del *router*, y a la hora de filtrar paquetes en las trazas con *Wireshark*.

3.4. Metodología

Más allá de las tecnologías utilizadas, es necesario hablar sobre la metodología de la ingeniería de software utilizada. Para el desarrollo de este proyecto se ha utilizado una metodología ágil, inspirada en *Scrum*, pero alterada para poder adaptarse a las especificaciones de este trabajo, haciendo especial hincapié en que no se contaba con un equipo de desarrollo. La metodología se basaría en iteraciones, las cuales constarían de varias fases:

- **Fase 1**: Diseño del protocolo inicial o rediseño del mismo bajo las conclusiones sacadas de la iteración anterior. Esta fase dio como resultado el diseño final de MUSE-RP mostrado en "Estudio y diseño de protocolos RUDP orientados a videojuegos"[12].
- **Fase 2**: Elección de metas prioritarias a desarrollar.
- **Fase 3**: Implementación de dichas metas prioritarias.
- **Fase 4**: *Testing* y pruebas de dichas metas, anotando todos los impedimentos que obstaculicen el desarrollo y posibles mejoras de diseño.
- Vuelta a la fase 1 teniendo en cuenta la información recogida en la fase 4.

Se llevaron a cabo, en total, 4 iteraciones importantes, desgranadas y explicadas en el apartado 4. Implementación de MUSE-RP.

4. Implementación de MUSE-RP

A continuación se describirán las diferentes iteraciones que hicieron falta para la implementación del protocolo RUDP MUSE-RP, con un apartado final hablando de los resultados de dicha implementación.

4.1. Primera iteración

La primera iteración del protocolo MUSE-RP se realizó, aproximadamente, entre el 18 de enero y el 11 de febrero de 2022, y estos fueron los pasos que dieron:

- Creación del proyecto .NET destinado a ser compilado como una librería.
- Creación de la base de las primeras clases diseñadas: *Host, Client, Server, ChannelHandler, ReliableChannelHandler, PartialReliableChannelHandler, MuseBuffer, MessageObject*.
- Implementación de un primer buffer que utilizaba una *Queue* para el almacenamiento de información. Este buffer es *thread safe*, es decir, cada método está protegido por un objeto de clase *mutex* que evita posibles fallos de concurrencia.
- Implementación casi completa de las clases *Host* y *MessageObject*, y primera iteración de las clases *Client, Server* y *ChannelHandler*.
- Implementación del algoritmo fiable y de *ReliableChannelHandler*.
- Comienzo de pruebas del algoritmo fiable.
- Implementación de temporizadores de reenvío usando hilos.
- Procesado de los mensajes de capa de aplicación en el hilo principal por parte de los *Hosts*.
- Primeras pruebas de concepto del mecanismo de inicio de conexión.

Tras esta primera iteración surgieron varios problemas no contemplados en el diseño, aparte de algunos errores en cuanto a la implementación de algunas de sus características. A raíz de ello se hizo un rediseño del protocolo, cuyos cambios se explicarán en el apartado siguiente.

4.2. Segunda iteración

La segunda iteración se realizó entre el 11 de febrero y el 2 de marzo, aproximadamente, y estuvo marcada por los siguientes hitos:

- Creación de una nueva clase encargada del manejo de *sockets*, que hará de intermediaria entre las clases *Host* y los canales (*ChannelHandler*). Este fue el primer fruto del desarrollo por iteraciones explicado en 3.4. Metodología. A la hora de implementar los *Host* y los canales de comunicación hubo problemas relacionados con la asignación de los *socket* a cada canal, sobre todo en el caso del servidor ya que tendrá muchas conexiones abiertas en los mismos canales. Es por ello que no tenía sentido seguir con esa arquitectura, y se añadió una sola clase que hiciera de intermediaria y que pudiese direccionar los paquetes recibidos en un socket al canal correspondiente.
- Creación de un hilo de procesado de mensajes en los *Host*. El uso de dicho hilo sería opcional, pudiendo acceder directamente al buffer de mensajes recibidos.
- Primera iteración de la implementación del algoritmo parcialmente fiable en la clase *ReliableChannelHandler.*
- Implementación funcional del mecanismo de inicio de conexión (mensajes INIT), que, dadas las dificultades que surgieron de la iteración anterior, se decidió modificar. Al principio solo se realizaba un *handshake* para los dos canales de comunicación. Sin embargo eso a veces causaba problemas en los *router NAT* (los más comunes en el uso personal), ya que los puertos de salida a veces no eran los especificados por punto terminal, al que le hacía pantalla.
- Creación de un segundo tipo de buffer, el *SortedMuseBuffer*, implementado sobre una lista ordenada (*SortedList).* La necesidad de este tipo de almacenamiento se hizo latente al implementar los algoritmos fiables, ya que a la hora de procesar el buffer de recibidos estos deben estar ordenados por número de secuencia.
- Se descarta una posible implementación de un canal no fiable. Por falta de tiempo, se tuvo que dejar la idea de crear un protocolo no fiable en cuanto entrega, pero si en cuanto el orden de llegada. Este podía sustituir al canal

parcialmente fiable, y se usaría en los casos de que no se necesitase ningún tipo de fiabilidad en un determinado flujo de datos.

- Primeras pruebas sencillas con puntos terminales en diferentes redes y con el uso de IP públicas. De los resultados de estas pruebas surgieron nuevos problemas y propuestas de diseño.

A raíz de la evolución del estado del protocolo y de que varias de las funcionalidades ya estaban siendo implementadas, surgieron nuevas necesidades y nuevos problemas que debían solucionarse, muchas veces con un cambio en el diseño.

4.3. Tercera iteración

La penúltima iteración tuvo como resultado un protocolo prácticamente funcional, aunque con algunos fallos que debían ser abordados en iteraciones posteriores. Cuenta como tercera iteración el desarrollo realizado entre el 2 de marzo y el 18 de marzo, y se realizaron las siguientes modificaciones/implementaciones:

- Cambio en la implementación del *SortedMuseBuffer.* En vez de usar una *SortedList*, se pasó a utilizar un conjunto ordenado (*SortedSet*), ya que tiene mejor rendimiento en cuanto a las operaciones que se deseaban realizar.
- Implementación de los mensaje *End* de finalización de conexión.
- Implementación de un mecanismo propio de ping (que utilizará un canal no fiable) manejado por la clase *PingController.* Esta funcionalidad surge a raíz de los impedimentos surgidos de las pruebas con la clase *Ping* propia de C#[28]. Muchos de los puntos terminales no contestaban a dicho mecanismo, ya que utilizaba el protocolo ICMP, y muchos *router* no responden a este tipo de mensajes debido a su uso en ataques informáticos. [29]. Destacar que del envío de pings y del cálculo del RTT (aparte del cálculo de los temporizadores de reenvío) lo hacía exclusivamente el servidor, cosa que, posteriormente, se rechazó.
- Implementación de un mecanismo de cerrado de conexión en caso de no respuesta, o *timeout.* Este mecanismo solo lo podía llevar a cabo el servidor, ya que era el que se encargaba del envío de mensajes ping. Sin embargo esto

causaba problemas, ya que el cliente no tenía forma de averiguar si el servidor seguía online, pero si al revés.

- Cambio en cuanto a la implementación de los temporizadores de envío. Ya no se usarían hilos, si no que se usaría una clase con un nivel más de abstracción: la clase *Timer* del espacio de nombres *System.Timers* [30]. Esta clase ofrece la posibilidad de asignar un tiempo de intervalo y de asignar un método que se ejecutará al terminar dicho intervalo. Su utilización facilitó el desarrollo del protocolo.
- Creación de un método comparador de *MessageObject* personalizado, para su uso en el *SortedMuseBuffer*. La necesidad de este protocolo surge de las limitaciones que ofrece utilizar un número de secuencia incremental. Para entender este problema, hay que tener en cuenta que los números de secuencia utilizados son *uint* (enteros sin signo), que como indica la documentación [20] ocupan 4 bytes. Es decir, su máximo está en 2^{32}-1, que es 4.294.967.295. Aunque es raro que ocurra, no es imposible pensar que en algún momento los números de secuencia pueden llegar a dicho número, sobre todo teniendo en cuenta que, según el diseño[12], los números de secuencia iniciales son aleatorios y no comienzan en 0. Para la solución de este problema se creó una clase que implementase la interfaz genérica *Icomparer* sobre *MessageObject* (puede verse su implementación en el Algoritmo 1) que tuviese en cuenta el tamaño de ventana y el máximo valor que puede tomar un *uint*.

```
#region CustomComparer
2 referencias
public class MessageObjectComparer : IComparer<MessageObject>
{
    private readonly uint windowSize;
    1 referencia
    public MessageObjectComparer(uint windowSize)
    {
        this.windowSize = windowSize;
    }
    1 referencia
    public int Compare(MessageObject x, MessageObject y)
    {
        if ((uint.MaxValue - x.getSequenceNumber() < windowSize - 1) && (y.getSequenceNumber() < windowSize - 1))
        {
            return -1;
        }
        if ((uint.MaxValue - y.getSequenceNumber() < windowSize - 1) && (x.getSequenceNumber() < windowSize - 1))
        {
            return 1;
        }
        return x.getSequenceNumber().CompareTo(y.getSequenceNumber());
    }
}
#endregion
```

Algoritmo 1:Clase MessageObjectComparer, que implementa la interfaz IComparer para los objetos MessageObject

- Creación de programas de prueba más intensivos, con mensajes a mostrar por consola sobre el estado de la comunicación (ACKs recibidos, conexiones completadas o reenvíos). Una de estas pruebas consistía en que el servidor enviase cada pocos segundos un paquete al cliente. Gracias a esta prueba se detectó qué el canal se sobrecargaba fácilmente, y que a los pocos minutos el servidor cerraba la conexión por *timeout*. No se encontró el motivo específico de dicho comportamiento errático, pero gracias al rediseño que se le dio al apartado de temporizadores (con, por ejemplo, la incorporación de la clase *Timer*) no volvió a ocurrir.

Con este nuevo repaso al protocolo ya podía empezar a probarse en un entorno real, pero estas pruebas hicieron que surgieran nuevos cambios y problemas, por lo que se realizó una última iteración.

4.4. Iteración final

En la última iteración del desarrollo del protocolo MUSE-RP (18 de marzo en adelante), ya no quedaba mucha funcionalidad por implementar. Es por ello que esta última etapa de desarrollo se centró en eliminar posibles *bugs* y comportamientos erráticos, además de refinar más el código. Muchos de estos errores salieron tras comenzar las pruebas en un entorno real (el videojuego implementado). Estos son los puntos clave de esta iteración:

- Implementación de un *PingController* también en el cliente. Este cambio surge de la necesidad de que el cliente también tenga constancia del estado del servidor, y que se desconecte de él en caso de que este no responda. A diferencia del servidor, que cuenta con un *PingController* por cada cliente, la clase *Client* solo necesitará uno.
- Revisión y solución de posibles problemas de concurrencia derivada del uso de hilos. Actualmente, MUSE-RP usa, como mínimo, 5 hilos por host: un hilo de procesado de mensajes en capa superior; dos hilos de recepción de paquetes, uno por cada socket usado; y dos hilos de procesado de paquetes, uno por cada socket usado. Aparte, por cada conexión se añaden 2 hilos más

de envío de paquetes, uno por cada canal. En resumen: un cliente tendrá 7 hilos activos mientras esté conectado al servidor, y el server tendrá *5 + 2* número de clientes conectados*. Se quiso aprovechar el potencial multihilo de los procesadores y las herramientas que proporciona el entorno de desarrollo elegido. Sin embargo, esto le añade una dificultad añadida, y es la posibilidad de que este paralelismo desemboque en fallos de concurrencia no contemplados. Esto se solucionó utilizando estructuras de datos concurrentes (tales como *ConcurrentDictionary)* incorporadas en el *framework* de .NET; o con el uso de *mutex* al usar variables compartidas.

- Creación de excepciones personalizadas y adaptadas a los posibles problemas que se puedan tener. Se crearon dos excepciones: *IncorrectMessageFormatException,* que se disparará cuando no se pueda transformar un paquete UDP a un paquete MUSE-RP; y *OversizedMessageDataException*, que saltará cuando se intente encapsular un paquete que supere el tamaño máximo del mensaje (actualmente de 1500 bytes, ya que, como se descubrió en la investigación previa, eso favorece a la baja latencia). Cuando se capturan este tipo de excepciones, se cierra el canal, la conexión o el host.

Con estos cambios, se consiguió lanzar la primera versión del protocolo MUSE-RP, lo cual no descarta futuras actualizaciones solucionando errores nuevos o añadiendo funcionalidades nuevas. Para la consulta del código de MUSE-RP visitar: https://github.com/Celtia-Martin/MUSE-RP-RUDP-protocol

4.5. Pruebas y resultados

A lo largo del desarrollo de este proyecto se han hecho varias pruebas de funcionalidad (previas a las pruebas con el videojuego). Gracias a estas pruebas pudieron encontrarse errores, tanto en el diseño como en la implementación. En el 9. Anexo: Código y salidas por pantalla de las pruebas con MUSE-RP pueden verse la dos pruebas más importantes que se realizaron. Una consiste en un chat muy simple con los demás jugadores, mientras que la otra consistía en un envío masivo de paquetes por parte del servidor a los demás clientes. Las salidas por pantalla de estas pruebas pueden verse también en 9. Anexo: Código y salidas por pantalla de las pruebas con MUSE-RP.

También se realizó una comprobación de la calidad del código con la herramienta *SonarCloud*, cuyos resultados pueden verse en la Ilustración 5. Con estos resultados, podemos concluir que el código es de calidad, ya que los bugs, *code smells* y brechas de seguridad detectadas no son graves.

Para los resultados relacionados con su comportamiento dentro de un entorno real, consultar 6.5 MUSE-RP.

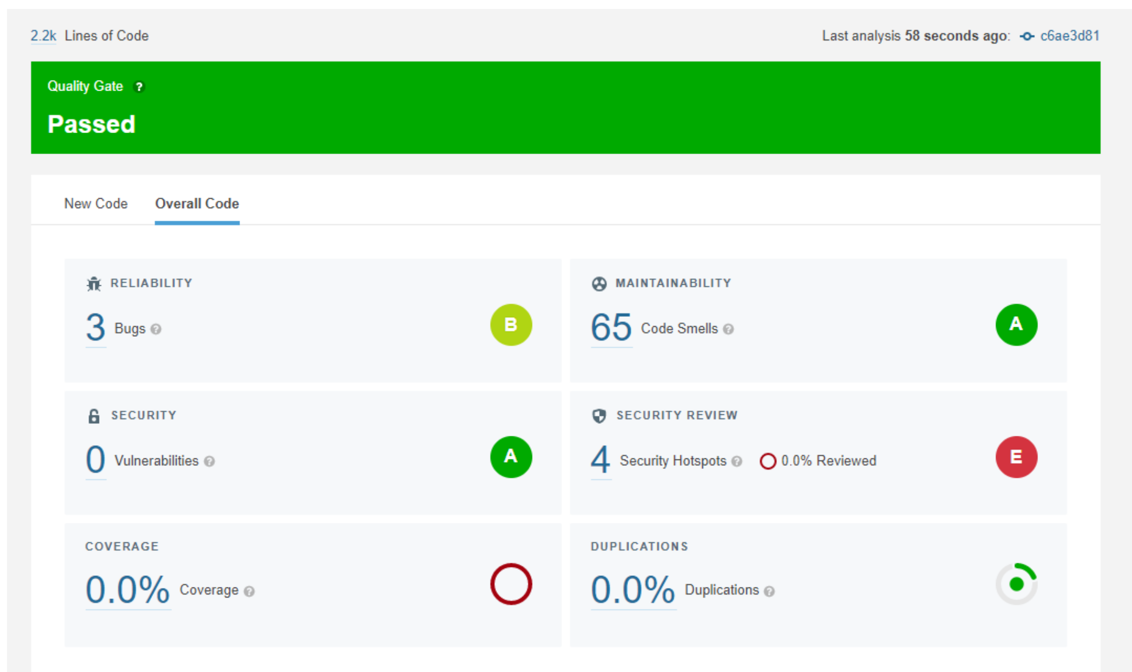

Ilustración 5: Resultados del análisis de código con SonarCloud para MUSE-RP

5.Diseño e implementación del juego de prueba

Para poder entrar mejor en el contexto de las pruebas del protocolo, se va a explicar a grandes rasgos el diseño del juego que se ha implementado para este cometido.

5.1. Diseño del gameplay del videojuego de prueba

No se quiso diseñar un juego muy complejo, ya que lo importante era tener una aplicación con la funcionalidad suficiente como para probar los diferentes protocolos estudiados (aparte de MUSE-RP). Sin embargo, se quiso que fuese algo dinámico, para poder tener datos empíricos sobre cómo se comportaban dichos protocolos. Para describir el juego, véase un resumen de en qué consiste el bucle de juego centrado en la jugabilidad:

- Primero aparecerá una pantalla en la que el jugador podrá elegir entre si crear una *room* (es decir, crear una partida donde él sea el servidor) o unirse a una ya creada. Si elige la primera opción, le aparecerá una pantalla con información sobre su IP y el puerto que va a ser utilizado (en el caso de utilizar dos (el caso de MUSE-RP) mostrará el fiable, ya que es al que tiene que conectarse el cliente inicialmente. Si elige la segunda opción, el usuario deberá introducir la IP y puerto del servidor. Esta IP puede ser pública, privada (si se está en la misma red que el servidor), o incluso proporcionada por softwares de creación de redes virtuales, como *Hamachi.*
- Una vez dentro de una *room*, el servidor podrá dar a comenzar partida en cualquier momento (habrá un botón para ello). Habrá una consola donde se anuncie el estado del punto terminal, por ejemplo si se está conectando, si se ha desconectado, o si hay nuevos clientes conectados.
- Una vez comenzada la partida cada jugador podrá controlar una cabeza con un color diferente y aleatorio, proporcionado por el servidor. Se moverá usando el ratón del ordenador, ya que esta cabeza sustituirá al cursor del ratón. Está cabeza tendrá una flecha que va a ir girando alrededor suyo, y pulsando el botón izquierdo del ratón, los jugadores podrán disparar (siempre y cuando se haya cumplido un tiempo entre disparo y disparo). La bala irá en la dirección en la que la flecha apuntaba en el momento del disparo.

- A lo largo de la partida irán apareciendo monstruos en pantalla. Si los jugadores los disparan, estos les darán puntos. Los puntos podrán verse sobre su cabeza.
- Si un jugador dispara a otro, el otro perderá unos cuantos puntos, pero el atacante no ganará puntos adicionales.
- Las partidas duran 3 minutos. Una vez terminado este tiempo, se mostrará una pantalla con los resultados de los diferentes jugadores y un botón que llevará a los usuarios al menú principal.

En la Ilustración 6 puede verse el paso por las diferentes pantallas del juego. Para un vistazo más técnico, ver la Ilustración 7 con un resumen del diagrama de clases de la aplicación. Destacar que todas las clases mostradas heredan de la clase de *Unity* llamada *MonoBehaviour,* clase que implementan los scripts asociados a objetos incorporados en la escena del juego. Para una lectura más limpia del diagrama, se han eliminado todos los métodos relacionados con esta herencia, además de otro tipo de métodos auxiliares, como los manejadores específicos de cada tipo mensaje o de conexión-desconexión, por ejemplo.

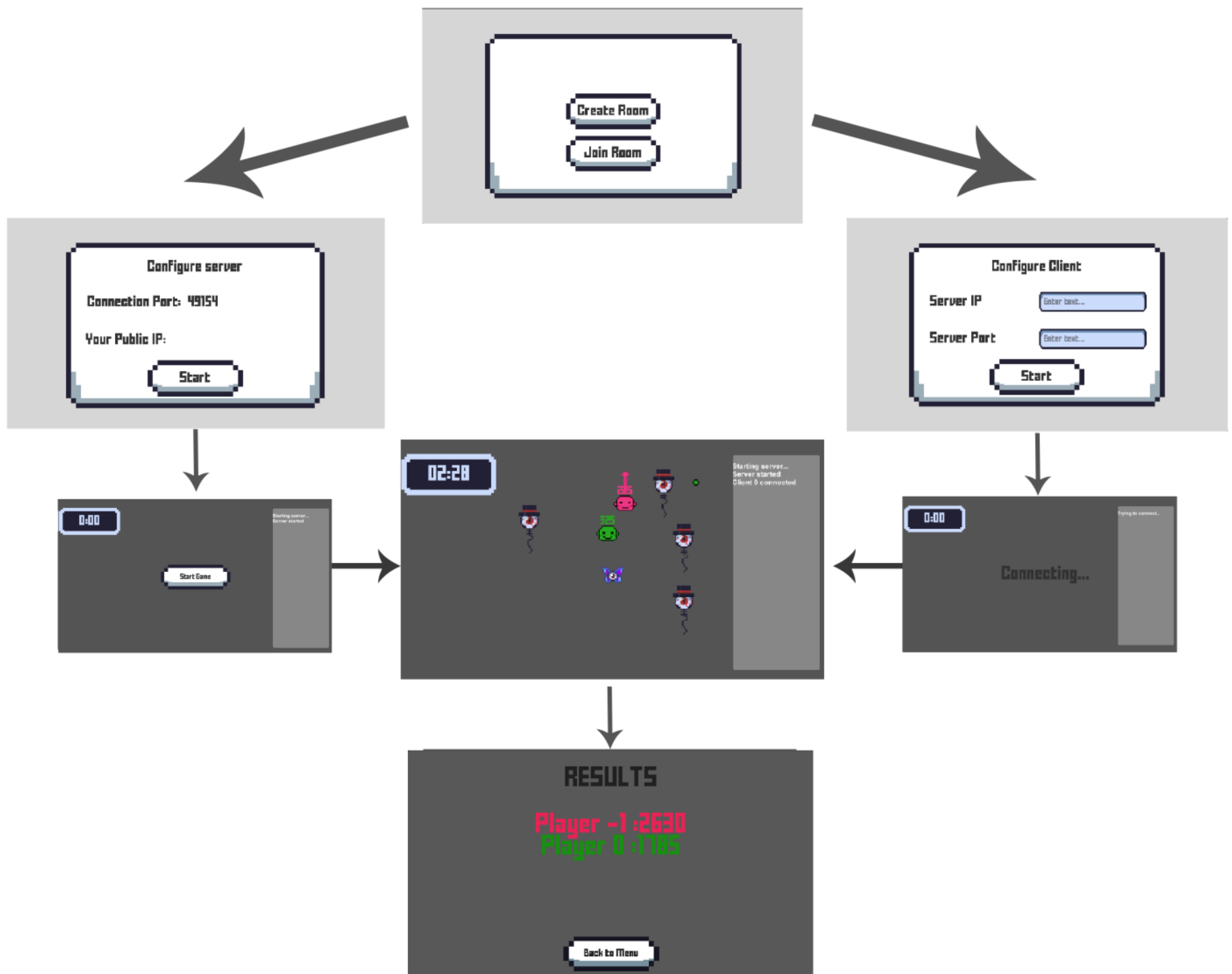

Ilustración 6: Paso por pantallas en el videojuego creado para las pruebas de los diferentes protocolos

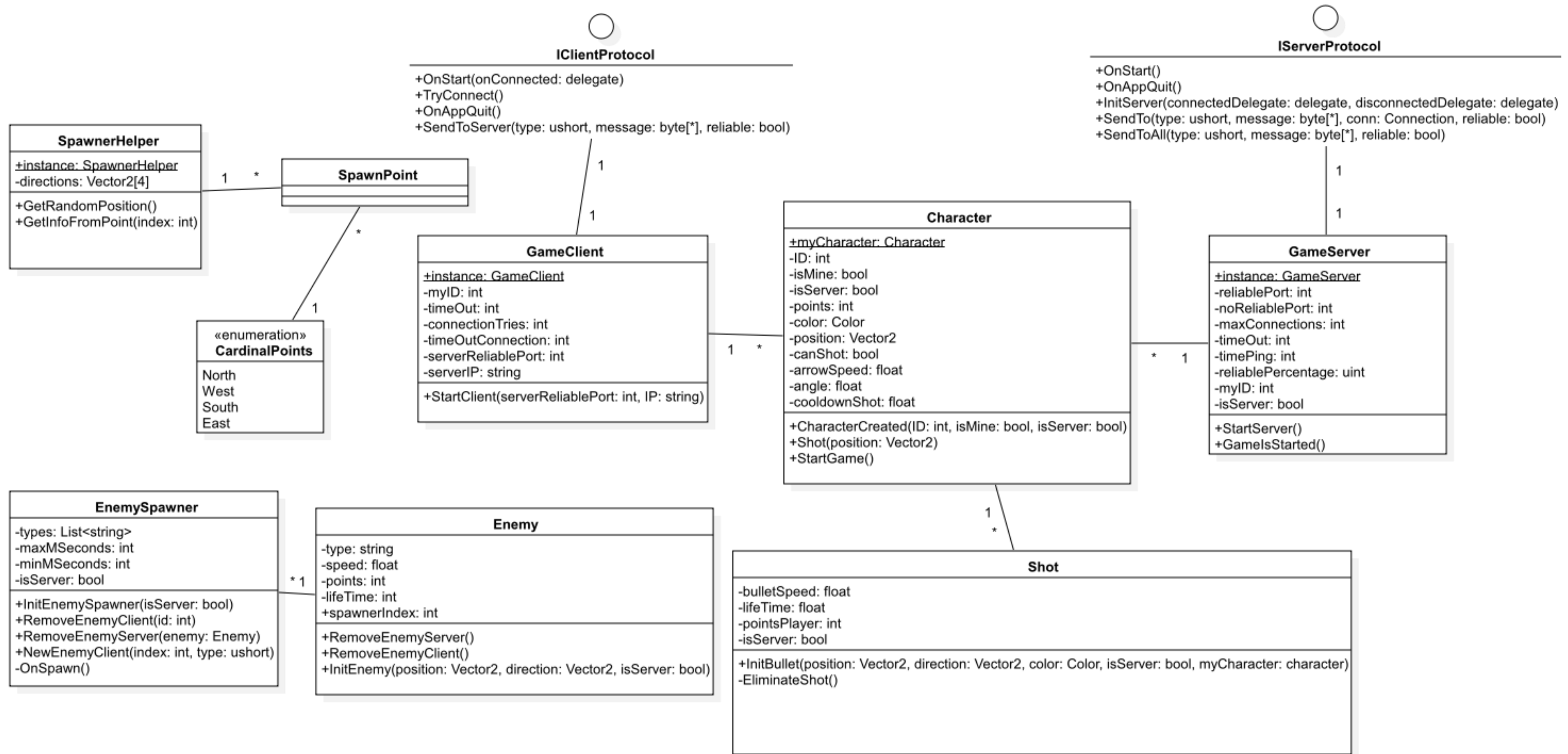

Ilustración 7: Diagrama UML con las clases principales del juego de prueba, junto a sus atributos y métodos más destacables.

5.2. Diseño de las comunicaciones del juego de prueba

Se quiso implementar una arquitectura Cliente-Servidor, ya que, como se ha hablado, es la más cómoda de utilizar. Además, MUSE-RP y muchos otros protocolos están especializados en ella. Se concretó, además, que el servidor sería también un jugador. Es decir, un jugador deberá crear una partida en modo servidor, y los demás deberán unirse a ella. Además, se quiso aspirar a la creación de un servidor autoritativo. Esto implica que será la ejecución del jugador que hace de servidor la que se encargue de tareas tales como:

- Hacer aparecer y desparecer los enemigos en posiciones aleatorias.
- Mantener información sobre la puntuación de los jugadores.
- Detectar las colisiones entre los enemigos y los disparos.
- Asignar los identificadores y colores de cada jugador.
- Generar los disparos cuando un jugador activa el comando de disparar

Para estas tareas se tuvieron que especificar todos los tipos de mensaje que podrían enviar y/o recibir los jugadores. Son los siguientes:

- **Mensaje tipo 4:** Nuevo jugador. Se recibe la información del color del nuevo jugador conectado junto a su ID.
- **Mensaje tipo 5:** Posición del jugador N. Se recibe un vector con la posición de un jugador determinado, junto a su identificador.
- **Mensaje tipo 6:** Un jugador se ha ido. Se recibe el identificador del jugador que se ha ido para borrarlo de la lista de jugadores.
- **Mensaje tipo 7:** Un jugador ha disparado. Se recibe la posición donde estaba la flecha en el momento del disparo y el jugador que ha disparado.
- **Mensaje tipo 8:** Se han añadido/quitado puntos al jugador N. Se recibe el incremento (ya sea positivo o negativo) de los puntos y qué jugador es el afectado.
- **Mensaje tipo 9:** Enemigo desaparecido. Se recibe el identificador del enemigo que ha desaparecido del área de juego.

- **Mensaje tipo 10:** Enemigo nuevo. Se recibe el punto de aparición y el tipo de enemigo nuevo.
- **Mensaje tipo 11:** Comienza el juego. Sirve para avisar de que la partida ha comenzado.
- **Mensaje tipo 12:** Termina el juego. Sirve para avisar de que la partida ha terminado.
- **Mensaje tipo 66:** Información del personaje del jugador. Se recibe el identificador y el color del personaje del jugador pertinente.

Se evitaron usar los números del 0 al 3 ya que algunos protocolos los usaban internamente para otro tipo de cometidos y daba fallos en la ejecución.

Para poder modificar la tecnología usada en la capa de transporte sin interferir en la arquitectura del juego, se crearon dos interfaces (*IClientProtocol* y *IServerProtocol)*, con los métodos comunes de los clientes y servidores creados con los diferentes protocolos. Estas interfaces definen funciones tales como métodos de envío, métodos de añadido de manejadores por tipo de mensaje, o métodos de inicio y parado de los puntos terminales. Por cada protocolo se crearon dos clases que implantaban estas interfaces.

El código de este juego de prueba puede encontrarse en: https://github.com/Celtia-Martin/MUSERP-Test

6. Comparativa de protocolos

En este apartado van a explorarse los resultados de la implementación de los diferentes protocolos en el juego de prueba. Se va a abordar desde 3 perspectivas diferentes: dos empíricas (comodidad de uso en cuanto a su desarrollo, y experiencia dentro del juego), y la información aportada por la captura de sus paquetes con *Wireshark*.

Todas las pruebas se realizaron en las mismas condiciones y entorno. Primero se creó una red virtual privada entre dos ordenadores en dos ubicaciones y redes diferentes (uno en Móstoles y otro en Palma de Mallorca). Las direcciones proporcionadas por dicha red virtual serían las utilizadas en la conexión entre ambas máquinas. Después de crear dicha conexión, se jugó una partida en *build* (es decir, con el juego compilado), de 3 minutos por cada protocolo probado. Es decir: los resultados obtenidos por cada protocolo son fruto de una partida completa de 3 minutos, y todas las capturas se hicieron en la máquina servidor. Que el entorno y las condiciones de prueba fuesen iguales fue clave para hacer una comparativa válida.

6.1. UDP

Para hacer un buen estudio sobre los protocolos RUDP, también era necesario probar el rendimiento de UDP sin protocolo.

6.1.1. Comodidad al desarrollar

Al no estar encapsulado en ningún protocolo, el desarrollo con UDP fue tosco y poco intuitivo. Aunque ha de decirse que el utilizar las interfaces creadas para hacer las pruebas ayudó en esta tarea.

Muchos de los mecanismos que incorporan los protocolos RUDP (como la creación de conexiones, o uso de manejadores por tipo de mensaje), tuvieron que implementarse vagamente para que el juego funcionase y el diseño se adaptase a la interfaz. Por ello, las clases que se usaron para implementar este protocolo tendrán estructuras de datos dedicadas a gestionar los manejadores (un diccionario) y las conexiones.

De hecho, al no estar orientado a conexión, tuvo que implementarse un manejador para un tipo de mensaje 0, que funcionaría como un mensaje de inicio de conexión. Lo que no se implementaron fueron mecanismos de desconexión, entre otros, ya que añadirle más capas de abstracción sería como crear un nuevo protocolo.

Configuración usada para las pruebas: UDP en sí no puede configurarse, sin embargo mencionar que se utilizó un buffer de 2000 bytes para la llegada de paquetes.

6.1.2. Experiencia empírica

Actualmente la red es muy segura, ya que ha estado evolucionando mucho con los años. Sin embargo, esto no impide que algunos paquetes se pierdan. En general la experiencia fue satisfactoria y la tasa de refresco de la información gráfica era decente, aunque el carácter no fiable del protocolo causó algunos errores en algunas ejecuciones. Esto fue debido a que se perdieron mensajes críticos. En la Ilustración 8 pueden verse los resultados erróneos fruto de una partida hecha sobre UDP. Una manera sencilla de solucionar, por ejemplo, el caso de las puntuaciones, sería que el servidor enviase la puntuación completa. Esto no se hizo así ya que precisamente se quería forzar una situación en la que pudiese comprobarse empíricamente que ha sucedido una pérdida crítica.

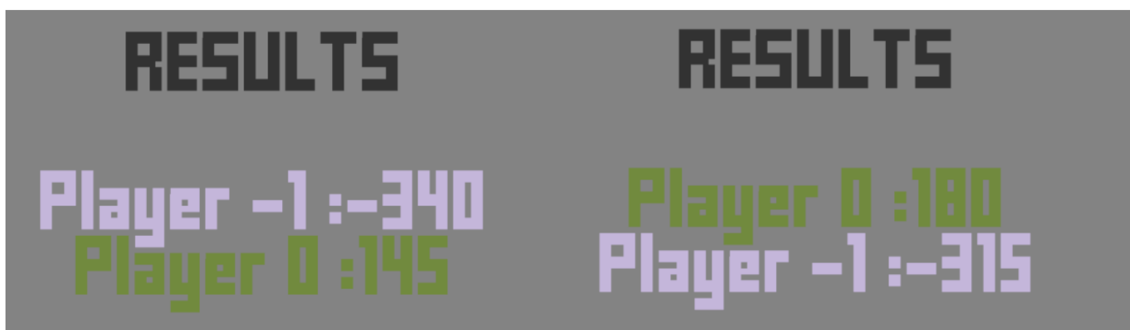

Ilustración 8: Puntuaciones erróneas producto de una partida al juego de prueba sobre UDP sin protocolo por encima

6.1.3. Trazas *Wireshark*

2592	50.530747	25.70.17.96	25.80.114.210	UDP	56	49154 → 53264 Len=14
2593	50.538318	25.80.114.210	25.70.17.96	UDP	60	53264 → 49154 Len=14
2594	50.544818	25.70.17.96	25.80.114.210	UDP	56	49154 → 53264 Len=14
2595	50.551573	25.70.17.96	25.80.114.210	UDP	56	49154 → 53264 Len=14
2596	50.561122	25.80.114.210	25.70.17.96	UDP	60	53264 → 49154 Len=14
2597	50.565383	25.70.17.96	25.80.114.210	UDP	56	49154 → 53264 Len=14
2598	50.571813	25.70.17.96	25.80.114.210	UDP	56	49154 → 53264 Len=14
2599	50.592915	25.70.17.96	25.80.114.210	UDP	50	49154 → 53264 Len=8
2600	50.600442	25.80.114.210	25.70.17.96	UDP	60	53264 → 49154 Len=14
2601	50.606863	25.70.17.96	25.80.114.210	UDP	56	49154 → 53264 Len=14
2602	50.620435	25.70.17.96	25.80.114.210	UDP	48	49154 → 53264 Len=6

Ilustración 9: Muestra de paquetes capturados usando UDP en el juego de prueba

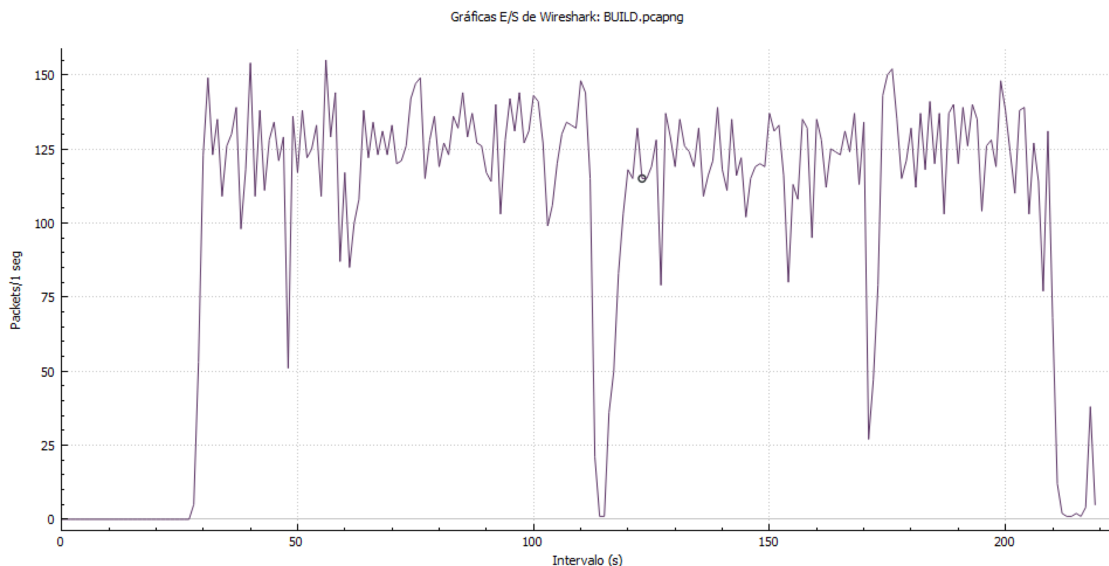

Ilustración 10: Gráfica de paquetes por segundo a lo largo de la comunicación con UDP

Para el análisis de los paquetes en *Wireshark* se utilizó el filtro *ip.addr==<<ip de hamachi de la otra máquina>>&&!icmp&&!ssdp*. Al haberse usado *Hamachi* pudieron analizarse los paquetes capturados solo en esa interfaz, lo que facilitó el análisis.

En la Ilustración 9 puede verse una pequeña muestra de los paquetes capturados. Puede apreciarse que el tamaño de paquete que más se repite es de 14 bytes. No es de extrañar ya que son paquetes de posición de los jugadores *(2 x tamaño float(4) + 2 x tamaño int (4) + tamaño ushort (2))*. También se puede observar que el servidor es la máquina que más mensajes envía. Esto no ocurrirá de ahora en adelante, porque todos los

demás protocolos implementan el mecanismo de reconocimiento de mensajes por ACK, por tanto habrá un tráfico de mensajes ACK respondiendo al servidor.

En la Ilustración 10 puede verse la gráfica generada por *Wireshark* sobre la cantidad de paquetes enviados por segundo en el intervalo de la conexión. Esta cifra rara vez ha superado los 150 paquetes por segundo en esta interacción. También puede verse en esta gráfica el transcurso del juego, como por ejemplo algunas franjas con pocos datos, ya que hubo problemas de conexión.

6.2 Telepathy

Telepathy es el protocolo orientado a videojuegos utilizado por encima de TCP que se ha usado para las comparativas con los demás protocolos. Ofrece bastantes herramientas que facilitaron la incorporación de TCP al videojuego de prueba.

6.2.1. Comodidad al desarrollar

TCP por si solo es difícil de manejar:

- No se puede controlar cuando se mandan los paquete
- TCP no trabaja con mensajes, si no que crea un flujo de datos. Esto añade una dificultad extra a la hora de dividir dicho *stream* para poder manejar los mensajes originales.
- Al igual que UDP, no cuenta con mecanismos para el manejo de mensajes por tipo, o métodos manejadores de conexión.

Es por ello que para facilitar el desarrollo de la prueba se hizo uso de una librería a un nivel aún más alto de abstracción, de la que ya se habló en 2.1.2 Protocolos probados.

El uso de esta librería aporto comodidad a la implementación del protocolo. Cuenta con métodos para añadir acciones en caso de recepción de datos, de conexión y de desconexión, lo que facilitó la incorporación al juego. Sin embargo, no cuenta con un mecanismo de manejadores de paquetes por tipo, y debe ejecutarse un método *Tick(int processLimit)* por cada llamada al método *Update* del bucle de juego incorporado en *Unity*.

Configuración usada para las pruebas: TCP no se puede configurar, pero *Telepathy* sí, y esta es la configuración que se utilizó:

- Máximo tamaño de mensaje: 2000 bytes
- Conexiones máximas: 20 clientes
- *NoDelay* activado, desactiva el algoritmo de *Nagle* que fusiona paquetes, bajando la latencia de CPU pero aumentando el ancho de banda.

6.2.2. Experiencia empírica

Al tratarse de TCP no hubo errores críticos en el flujo del juego, como si ocurrió en el caso de UDP. Sin embargo, hubo momentos de *lag* y cortes, es decir, a veces el jugador que actuaba como cliente veía su experiencia perjudicada por que no se procesaban los mensajes correspondientes a la posición con la suficiente tasa de refresco. De hecho, como puede verse en la captura y en la Ilustración 11, hubo varios momentos en los que hubo pérdida de paquetes y se necesitaron varias retrasmisiones. Al tratarse, además, de paquetes de posición, su pérdida no hubiera sido gran problema, y quizás el retraso creado por la espera del paquete perdido fue más perjudicial para la experiencia del usuario, comparado con la perdida de unos pocos paquetes. Sin embargo, en general la experiencia fue satisfactoria.

6.2.3. Trazas *Wireshark*

```
5800 55.831368 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=26659 Ack=55707 Win=64840 Len=18
5801 55.831433 25.80.114.210    25.70.17.96      TCP    82 [TCP Dup ACK 5739#21] 63269 → 49154 [ACK] Seq=26677 Ack=55707 Win=64840 Len=0 SLE=56013 SRE=56153 S
5802 55.834930 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=56189 Ack=26677 Win=64914 Len=18
5803 55.840668 25.80.114.210    25.70.17.96      TCP    66 63269 → 49154 [ACK] Seq=26677 Ack=56153 Win=64394 Len=0 SLE=55743 SRE=55797
5804 55.841560 25.70.17.96      25.80.114.210    TCP    66 49154 → 63269 [PSH, ACK] Seq=56207 Ack=26677 Win=64914 Len=12
5805 55.841604 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=56219 Ack=26677 Win=64914 Len=18
5806 55.844120 25.80.114.210    25.70.17.96      TCP    60 63269 → 49154 [ACK] Seq=26677 Ack=56189 Win=64358 Len=0
5807 55.844815 25.80.114.210    25.70.17.96      TCP    66 [TCP Dup ACK 5806#1] 63269 → 49154 [ACK] Seq=26677 Ack=56189 Win=64358 Len=0 SLE=55707 SRE=55743
5808 55.845313 25.80.114.210    25.70.17.96      TCP    66 [TCP Dup ACK 5806#2] 63269 → 49154 [ACK] Seq=26677 Ack=56189 Win=64358 Len=0 SLE=55797 SRE=55815
5809 55.845326 25.80.114.210    25.70.17.96      TCP    66 [TCP Dup ACK 5806#3] 63269 → 49154 [ACK] Seq=26677 Ack=56189 Win=64358 Len=0 SLE=55995 SRE=56013
5810 55.845343 25.70.17.96      25.80.114.210    TCP   102 [TCP Fast Retransmission] 49154 → 63269 [PSH, ACK] Seq=56189 Ack=26677 Win=64914 Len=48
5811 55.849039 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=26677 Ack=56189 Win=64358 Len=18
5812 55.862269 25.70.17.96      25.80.114.210    TCP    90 49154 → 63269 [PSH, ACK] Seq=56237 Ack=26695 Win=64896 Len=36
5813 55.863530 25.80.114.210    25.70.17.96      TCP    60 63269 → 49154 [ACK] Seq=26695 Ack=56219 Win=64328 Len=0
5814 55.869881 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=26695 Ack=56237 Win=64310 Len=18
5815 55.871077 25.80.114.210    25.70.17.96      TCP    66 [TCP Dup ACK 5814#1] 63269 → 49154 [ACK] Seq=26713 Ack=56237 Win=64310 Len=0 SLE=56189 SRE=56237
```

Ilustración 11: Muestra de paquetes TCP en la prueba del uso de Telepathy del videojuego con recepción de ACK duplicados y retrasmisión rápida

```
6600 59.916788 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63771 Ack=30295 Win=65418 Len=18
6601 59.929616 25.80.114.210    25.70.17.96      TCP    60 63269 → 49154 [ACK] Seq=30295 Ack=63753 Win=65098 Len=0
6602 59.938117 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63789 Ack=30295 Win=65418 Len=18
6603 59.941649 25.80.114.210    25.70.17.96      TCP    60 63269 → 49154 [ACK] Seq=30295 Ack=63789 Win=65062 Len=0
6604 59.958758 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63807 Ack=30295 Win=65418 Len=18
6605 59.968776 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=30295 Ack=63807 Win=65044 Len=18
6606 59.979628 25.70.17.96      25.80.114.210    TCP    90 49154 → 63269 [PSH, ACK] Seq=63825 Ack=30313 Win=65400 Len=36
6607 59.988394 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=30313 Ack=63825 Win=65026 Len=18
6608 60.000737 25.70.17.96      25.80.114.210    TCP    90 49154 → 63269 [PSH, ACK] Seq=63861 Ack=30331 Win=65382 Len=36
6609 60.009627 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=30331 Ack=63861 Win=64990 Len=18
6610 60.014536 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63897 Ack=30349 Win=65364 Len=18
6612 60.021009 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63915 Ack=30349 Win=65364 Len=18
6613 60.023593 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=30349 Ack=63861 Win=64990 Len=18
6614 60.035217 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63933 Ack=30367 Win=65346 Len=18
6615 60.035297 25.70.17.96      25.80.114.210    TCP    72 49154 → 63269 [PSH, ACK] Seq=63951 Ack=30367 Win=65346 Len=18
6616 60.043066 25.80.114.210    25.70.17.96      TCP    60 63269 → 49154 [ACK] Seq=30367 Ack=63915 Win=64936 Len=0
6617 60.043701 25.80.114.210    25.70.17.96      TCP    72 63269 → 49154 [PSH, ACK] Seq=30367 Ack=63915 Win=64936 Len=18
6618 60.056655 25.70.17.96      25.80.114.210    TCP    90 49154 → 63269 [PSH, ACK] Seq=63969 Ack=30385 Win=65328 Len=36
```

Ilustración 12: Muestra de paquetes TCP en la prueba del uso de Telapathy del videojuego

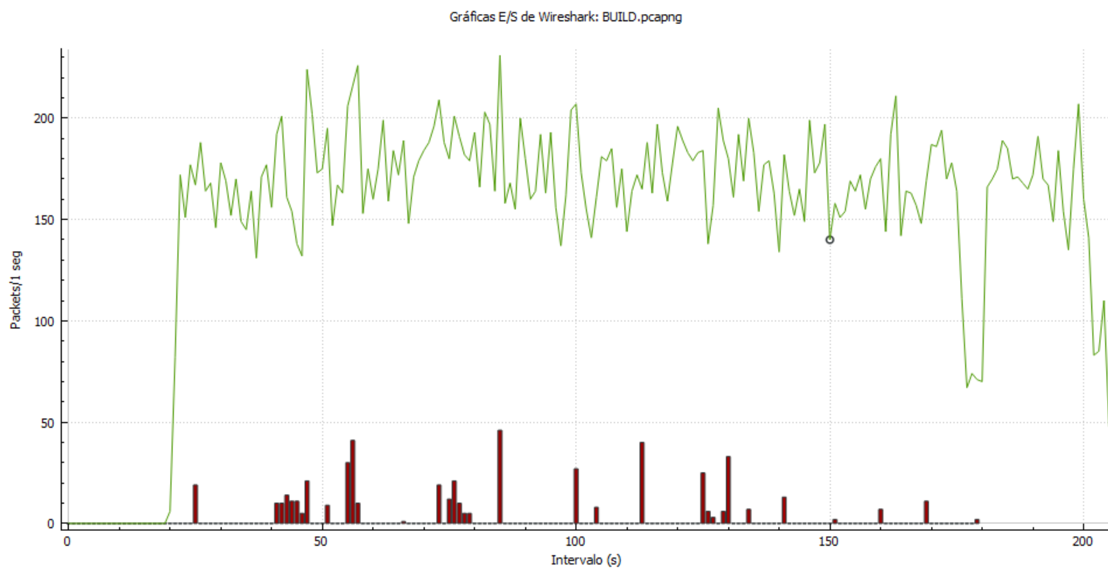

Ilustración 13: Gráfica de flujo de paquetes por segundo y errores de TCP a lo largo de la comunicación con TCP

En el caso de *Telepathy*, la medida de paquetes por segundo ha sido más regular, rondando los 175, pero con picos cercanos a 250. Esto puede verse en la Ilustración 13. Que sea más estable y mayor a UDP puede deberse a que se incorporan al tráfico los mensajes ACK. Estos, aunque no contienen carga útil, pueden llegar a duplicar el número de paquetes enviados. En la Ilustración 12 puede verse como estos paquetes ACK

aparecen de vez en cuando para indicarle al emisor que la información ha llegado correctamente.

La Ilustración 13, además, incorpora información sobre el número de errores de TCP que ha habido a lo largo de la comunicación (las barras oscuras). Como puede apreciarse, ha habido paquetes perdidos, con sus respectivas retrasmisiones, a la largo de toda la prueba, aunque por lo general no ha tenido consecuencias graves (más allá de perjudicar al *gamefeel*).

6.3 Ruffles

Ruffles es el primer protocolo RUDP cuyo desempeño se analiza en este documento. Tras el análisis de TCP y UDP, será interesante explorar cómo se comporta la alternativa catalogada como híbrida entre estos dos protocolos.

6.3.1. Comodidad al desarrollar

Ruffles es uno de los protocolos RUDP más completos y configurables que se han encontrado en la investigación previa al desarrollo de MUSE-RP. En los demás protocolos no se pudo configurar un modo fiable y otro no fiable (como si es el caso de MUSE-RP), pero en *Ruffles* sí que se puede configurar el canal utilizado, por tanto se usó el fiable para los mensajes críticos y el no fiable (pero ordenado) para los demás(como es el caso de la posición). Gracias al uso de la clase *SocketConfig*, la personalización del uso de este protocolo fue sencilla y agradable.

Sin embargo, para ir manejando cada mensaje hizo falta implementar un bucle de escucha. Usando el método *Poll()* de *RuffleSocket* se extrae un *NetworkEvent*, que puede ser de varios tipos: datos, comienzo de conexión, fin de conexión, entre otros. Según el tipo, pueden realizarse diferentes acciones. La necesidad de implementar este método aumenta la flexibilidad del desarrollo en cuanto a adaptar el protocolo a sus necesidades, pero también puede resultar tedioso. En el Algoritmo 2 puede verse la creación de este hilo de escucha para el cliente.

Configuración usada para las pruebas:

- Uso de varios tipos de canales (fiable y no fiable)
- Uso de simulador: desactivado.
- *TimeOut* activado y de 30 segundos
- Fusión de paquetes desactivada

```
private void ListeningThread()
{
    while (clientSocket.IsRunning)
    {
        NetworkEvent clientEvent = clientSocket.Poll();
        switch (clientEvent.Type)
        {
            case NetworkEventType.Nothing:
                break;
            case NetworkEventType.Connect:
                serverConnection = clientEvent.Connection;
                onConnected?.Invoke();
                break;
            case NetworkEventType.Disconnect:
                onDisconnected?.Invoke();
                break;
            case NetworkEventType.Timeout:
                onDisconnected?.Invoke();
                break;
            case NetworkEventType.Data:
                ushort type = BitConverter.ToUInt16(clientEvent.Data.Array, 0);
                Debug.Log("Type of message " + type);
                if (handlerDictionary.TryGetValue(type, out Action<byte[]> value))
                {
                    value?.Invoke(clientEvent.Data.Take(clientEvent.Data.Count).ToArray());
                }
                break;
            case NetworkEventType.UnconnectedData:
                break;
            case NetworkEventType.BroadcastData:
                break;
            case NetworkEventType.AckNotification:
                break;
        }
        clientEvent.Recycle();
    }
}
```

Algoritmo 2:Bucle de escucha de mensajes implementado para el cliente Ruffles

6.3.2. Experiencia empírica

Ruffles se adaptó satisfactoriamente al flujo del juego. Salvo escasos episodios de bajada de FPS (*frames* por segundo), la partida realizada con este protocolo fue exitosa y la inmersión no se vio perjudicada. Además de esto, al poder usar un canal fiable, los paquetes críticos no sufrían peligro de extraviarse. En definitiva, proporcionó una buena experiencia de juego.

6.3.3. Trazas *Wireshark*

4636	104.728…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4637	104.728…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4638	104.733…	25.80.114.210	25.70.17.96	UDP	60	61521 → 49154 Len=18
4639	104.735…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4640	104.742…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4641	104.748…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4642	104.757…	25.80.114.210	25.70.17.96	UDP	60	61521 → 49154 Len=18
4643	104.757…	25.80.114.210	25.70.17.96	UDP	60	61521 → 49154 Len=12
4644	104.762…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4645	104.769…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4646	104.774…	25.80.114.210	25.70.17.96	UDP	60	61521 → 49154 Len=18
4647	104.776…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4648	104.790…	25.70.17.96	25.80.114.210	UDP	60	49154 → 61521 Len=18
4649	104.795…	25.80.114.210	25.70.17.96	UDP	60	61521 → 49154 Len=18

Ilustración 14: Muestra de paquetes capturados en el transcurso de una partida usando Ruffles

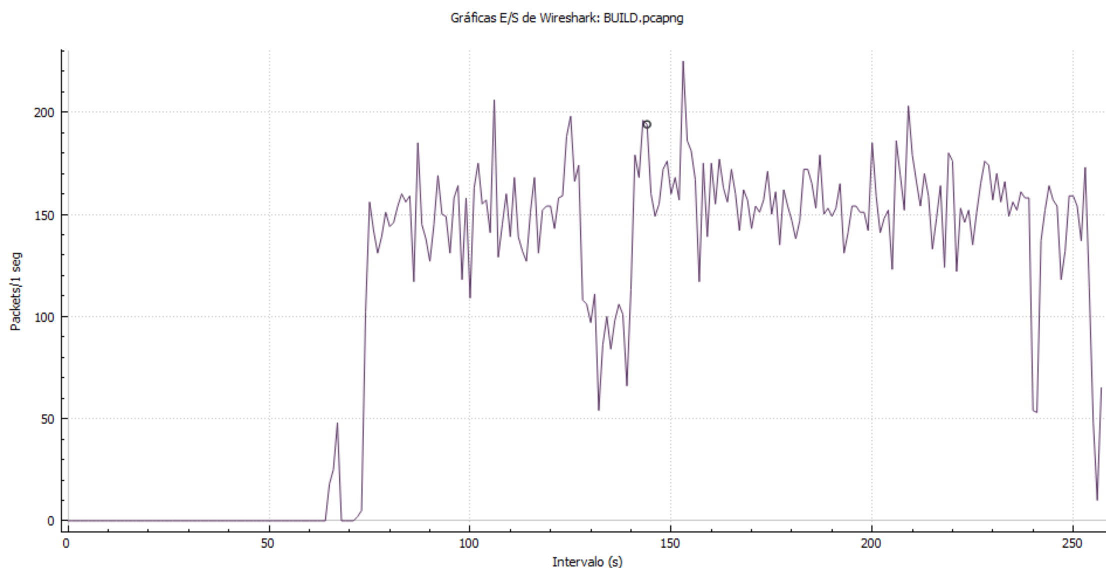

Ilustración 15: Gráfica de paquetes por segundo en el transcurso de una partida usando Ruffles

En la Ilustración 14 puede verse el flujo de unos cuantos paquetes de posición entre el cliente y el servidor *Ruffles* (los de 18 bytes). Puede apreciarse que el servidor es el que más paquetes envía, y no parece haber respuesta por parte del cliente. Esto es porque para este tipo de paquetes se usó un canal no fiable (pero que garantizaba la llegada en orden). Para los demás paquetes sí que se mandan ACKs, (los paquetes de 12 bytes) , que garantizan la entrega.

En la Ilustración 15 se ve, por otro lado, que los paquetes por segundo enviados en el flujo de la conversación ronda los 150-175 de media, con picos cercanos más allá de 200. Esta es una cifra mayor a la de UDP, pero menor que la de TCP.

6.4 GServer

GServer, a diferencia de *Ruffles*, es específico para videojuegos, según confirma el *README* del repositorio donde se aloja[17]. El estudio de su comportamiento será clave para poner a MUSE-RP a prueba contra un protocolo con el que comparte finalidad.

6.4.1. Comodidad al desarrollar

Trabajar con *GServer* fue cómodo y no requirió de muchas líneas de código ni implementar funcionalidad por encima. Incorpora un mecanismo para añadir manejadores según el tipo de paquete, e incluso de manejo de excepciones. En general, fue un protocolo sencillo de incorporar .

Configuración usada para las pruebas: GServer no cuenta con mucha flexibilidad en cuanto a su configuración, pero sí que se puede configurar el modo de fiabildiad de los mensajes (que se dejó en fiable porque el modo no fiable estaba causando problemas en las pruebas), y el tiempo de tick (cada 10 milisegundos).

6.4.2. Experiencia empírica

En cuanto a experiencia del usuario, *GServer* no cumplió con las expectativas. Fue, junto a TCP, uno de los protocolos probados con la peor experiencia de usuario proporcionada. Esto fue debido a un bajo rendimiento y una tasa de refresco de información insuficiente (perjudicando el *gamefeel*). Estos problemas no se experimentaron en ningún otro protocolo de los probados. Mencionar, además, que la configuración utilizada fue la misma que se muestra en el ejemplo que se da en la página del repositorio [17].

6.4.3. Trazas *Wireshark*

383…	159.208…	25.80.114.210	25.70.17.96	UDP	144	7777 → 49154 Len=102
383…	159.212…	25.70.17.96	25.80.114.210	UDP	360	49154 → 7777 Len=318
383…	159.221…	25.80.114.210	25.70.17.96	UDP	90	7777 → 49154 Len=48
383…	159.222…	25.70.17.96	25.80.114.210	UDP	204	49154 → 7777 Len=162
383…	159.228…	25.80.114.210	25.70.17.96	UDP	150	7777 → 49154 Len=108
383…	159.232…	25.70.17.96	25.80.114.210	UDP	198	49154 → 7777 Len=156
383…	159.238…	25.80.114.210	25.70.17.96	UDP	90	7777 → 49154 Len=48
383…	159.242…	25.70.17.96	25.80.114.210	UDP	96	49154 → 7777 Len=54
383…	159.249…	25.80.114.210	25.70.17.96	UDP	90	7777 → 49154 Len=48
383…	159.252…	25.70.17.96	25.80.114.210	UDP	150	49154 → 7777 Len=108
383…	159.262…	25.70.17.96	25.80.114.210	UDP	150	49154 → 7777 Len=108
383…	159.263…	25.80.114.210	25.70.17.96	UDP	90	7777 → 49154 Len=48
383…	159.269…	25.80.114.210	25.70.17.96	UDP	90	7777 → 49154 Len=48
383…	159.272…	25.70.17.96	25.80.114.210	UDP	150	49154 → 7777 Len=108
383…	159.280…	25.80.114.210	25.70.17.96	UDP	96	7777 → 49154 Len=54
383…	159.282…	25.70.17.96	25.80.114.210	UDP	198	49154 → 7777 Len=156
383…	159.290…	25.80.114.210	25.70.17.96	UDP	90	7777 → 49154 Len=48
383…	159.292…	25.70.17.96	25.80.114.210	UDP	204	49154 → 7777 Len=162

Ilustración 16: Muestra de paquetes UDP usando GServer en el videojuego de prueba

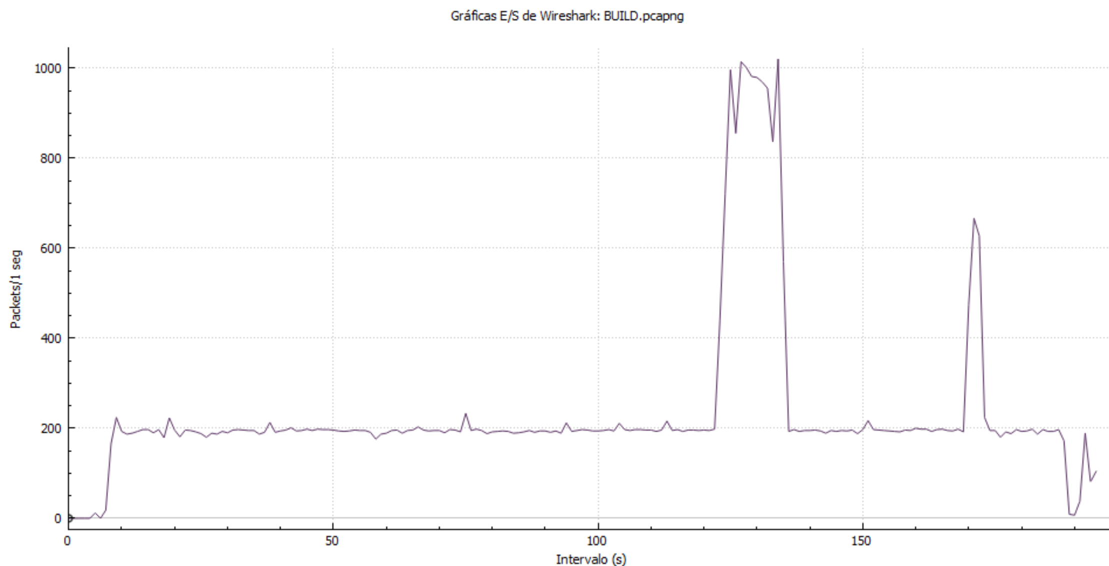

Ilustración 17: Gráfica de flujo de paquetes por segundo a lo largo de la comunicación con GServer

En la Ilustración 16 se ve una pequeña muestra de los paquetes capturados usando este protocolo. Lo primero que llama la atención (y que se tratará en más profundidad en 6.6 Comparativa general y estadísticas), es que el tamaño medio de paquete es mayor que en todos los casos anteriores (y que MUSE-RP, como se comprobará en el siguiente apartado). De hecho, los paquetes ACK, que no deberían tener mucha información, tienen 48 bytes de carga útil (en comparación con MUSE-RP, que son de 11 bytes, es un poco más del cuádruple).

Además, como puede apreciarse en la Ilustración 17, el flujo de paquetes por segundo ronda los 200 de media, con subidas hasta de caso 1000 paquetes por segundo. Este tipo de comportamiento no se ha visto en ningún otro protocolo, e indica que está haciendo un uso exagerado del ancho de banda. Esta medida, además, es mayor que los demás protocolos, lo que puede ser una razón de su bajo rendimiento a la hora de probar sus resultados empíricamente.

6.5 MUSE-RP

Ahora va a pasarse a analizar el comportamiento en un entorno real del protocolo RUDP diseñado [21], e implementado en este proyecto.

6.5.1. Comodidad al desarrollar

Como ya se ha explicado en 3.2. Diseño del protocolo RUDP implementado: MUSE-RP, MUSE-RP es altamente configurable, es decir, incorpora herramientas y funcionalidades personalizables para que el desarrollador adapte el protocolo a sus necesidades. Además se necesitan pocas líneas de código para arrancar el cliente o el servidor.

Sin embargo, un problema con el que no se contó es que *Unity* no es multihilo, y algunas operaciones (tales como destruir o cambiar la visibilidad de los objetos en la escena de juego), no pueden realizarse en otro sitio que no sea el hilo principal. Esto fue un problema ya que los *Host* de MUSE-RP tienen la opción de gestionar los mensajes en un hilo. Podían implementarse dos tipos de soluciones diferentes:

- Desactivar el procesado de mensajes en hilo y ejecutar *Host.ProcessNextMessage()* en algún tipo de método de los que usa *Unity* para actualizar el estado de la aplicación en el bucle de juego.
- Que los métodos manejadores no ejecuten la funcionalidad en sí, si no que añadan a una cola de acciones los métodos a ejecutar. Esta cola de acciones irá vaciándose y ejecutándose en los mismos métodos de los que se ha hablado en la otra posible solución.

Se optó por la segunda opción, para no tener que renunciar al carácter multihilo de MUSE-RP.

Configuración usada para las pruebas: MUSE-RP tiene bastantes cosas que configurar, por tanto se describe aquí lo que se ha usado para las pruebas:

- 20 conexiones máximas
- *TimeOut* a los 10 segundos.
- Ping cada 3 segundos
- Porcentaje de fiabilidad del 80% (es decir, 20% de pérdidas admitidas)
- Ventana de 1000 paquetes
- Tiempo de temporización inicial: 200 ms.

6.5.2. Experiencia empírica

En cuanto a la experiencia de juego, MUSE-RP pudo adaptarse perfectamente a su incorporación en un entorno real. El movimiento y los eventos de juego se comportaron de forma satisfactoria, apenas perjudicando a la inmersión del jugador. Hubo pocos episodios de cortes, o *lag*, en el flujo de juego, y tampoco hubo paquetes críticos perdidos. En general, puede compararse y competir contra los protocolos aquí mostrados sin problemas, llegando a superar el desempeño de TCP y de *GServer*, y aportádole fiabilidad a la eficiencia de UDP.

6.5.3. Trazas *Wireshark*

145…	124.011…	25.80.114.210	25.70.17.96	UDP	60	61032 → 49155 Len=11
145…	124.012…	25.80.114.210	25.70.17.96	UDP	65	61032 → 49155 Len=23
145…	124.013…	25.70.17.96	25.80.114.210	UDP	53	49155 → 61032 Len=11
145…	124.023…	25.70.17.96	25.80.114.210	UDP	65	49155 → 61032 Len=23
145…	124.023…	25.70.17.96	25.80.114.210	UDP	61	49154 → 50846 Len=19
145…	124.023…	25.70.17.96	25.80.114.210	UDP	65	49155 → 61032 Len=23
145…	124.047…	25.80.114.210	25.70.17.96	UDP	60	50846 → 49154 Len=11
145…	124.048…	25.80.114.210	25.70.17.96	UDP	60	61032 → 49155 Len=11
145…	124.048…	25.70.17.96	25.80.114.210	UDP	65	49155 → 61032 Len=23
145…	124.048…	25.80.114.210	25.70.17.96	UDP	60	61032 → 49155 Len=11
145…	124.048…	25.80.114.210	25.70.17.96	UDP	60	61032 → 49155 Len=11
145…	124.050…	25.80.114.210	25.70.17.96	UDP	60	61032 → 49155 Len=11
145…	124.050…	25.80.114.210	25.70.17.96	UDP	60	50846 → 49154 Len=11

Ilustración 18: Muestra de paquetes capturados usando MUSE-RP en el videojuego de prueba

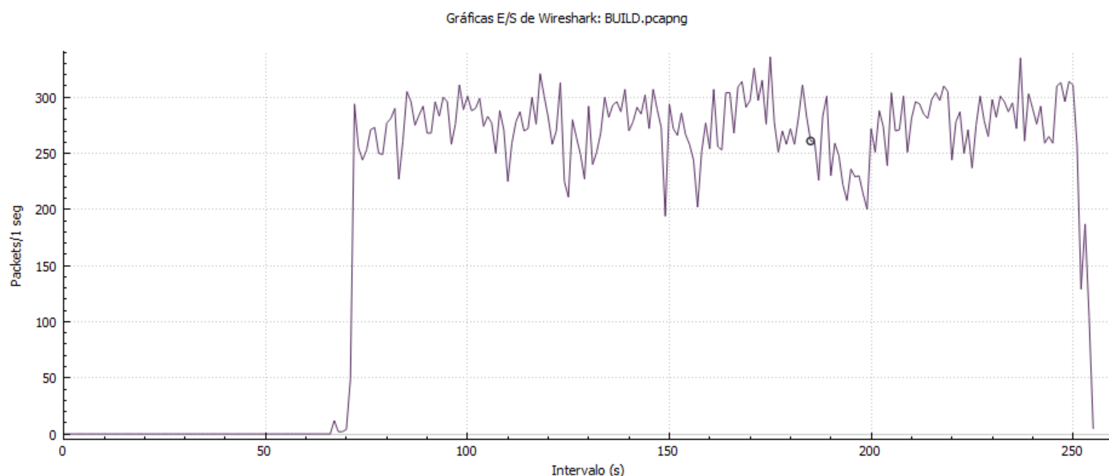

Ilustración 19: Gráfica de paquetes por segundo a lo largo de la partida usando MUSE-RP

En la Ilustración 18 puede verse una pequeña muestra de la captura de paquetes hecha jugando una partida usando MUSE-RP. Lo primero que resalta es el bajo tamaño de la carga útil de los paquetes, y la aparición de muchos paquetes ACK (los que tienen 11 bytes de carga útil, es decir, solo la cabecera). A diferencia de los demás protocolos fiables, que estarán usando algún mecanismo de fusión entre ACKs, MUSE-RP manda un ACK por cada mensaje recibido.

Esto también puede apreciarse en la Ilustración 19, ya que es el protocolo probado con más paquetes por segundo medio enviados (exceptuando el comportamiento anómalo de *GServer*), con alrededor de 250-300 paquetes por segundo de media.

En el siguiente apartado se hablará un poco más de otro tipo de estadísticas proporcionadas por *Wireshark*, y se podrá hacer una mejor comparación entre estos protocolos.

6.6 Comparativa general y estadísticas

Tabla 1: Estadísticas de los protocolos en el desempeño de una partida online en el videojuego de prueba

	UDP	**TCP(*Telepathy*)**	***Ruffles***	***GServer***	**MUSE-RP**
Tamaño medio de paquete	57,02 bytes	70,24 bytes	62,68 bytes	551,95 bytes	61,21 bytes
Porcentaje carga útil	24,1%	20,8%	32,3%	92,6%	27,5%
Bytes enviados	1,246 MB	2,182 MB	1,715 MB	25 MB	3,066 MB
Bytes por segundo enviados por el servidor	4,250 KB/s	6,500 KB/s	5,500 KB/s	75 KB/s	8 KB/s
Bytes por segundo enviados por el cliente	2,125 KB/s	5,125 KB/s	3,250 KB/s	58,250KB/s	8,125 KB/s
Paquetes enviados totales	21.856	31.074	27.379	45.710	50.107
Paquetes enviados por el servidor	14.906	16.671	17.311	23.560	25.060
Paquetes enviados por el cliente	6.950	14.403	10.068	22.150	25.047

En la Tabla 1 se resume algunos de los datos que ofrece *Wireshark,* referentes a la captura de paquetes que se ha hecho por cada protocolo en una partida al juego de prueba. Con esta tabla pueden compararse los diferentes protocolos en diversos aspectos de su comportamiento, y junto a la información empírica adquirida, pueden llevarse a cabo diversas conclusiones.

La primera de ellas es que, como no es de extrañar, el protocolo que menos ancho de banda consume es UDP. El tamaño de sus paquetes, al no contar con una cabecera adicional, es la más pequeña. También es el que menos datos ha mandado por segundo y en total. Además puede verse algo curioso, pero tampoco sorprendente: el servidor ha enviado un 114% más paquetes que el cliente. Esta diferencia no es tan grande en los demás protocolos (16% en TCP, 72% en *Ruffles*, 6,3% en *GServer*, prácticamente 0 en MUSE-RP), ya que todos los demás harán uso de mensajes ACK para el reconocimiento de mensajes, es decir, al tráfico normal de mensajes se le añade un tráfico de mensajes de

reconocimiento. *Ruffles* es el segundo protocolo con mayor diferencia entre paquetes enviados por el servidor y el cliente. Esto cumple con lo previsto, ya que los paquetes de posición, que son los más abundantes en este entorno, se mandan por un canal no fiable. Es decir, no necesita de mensajes de reconocimiento.

Otra cosa en la que uno se puede fijar es en que *GServer* (menos en la cantidad de mensajes enviados), es el que tiene los números más grandes. En el flujo de información envío un 715% más bytes que MUSE-RP*,* el segundo con más bytes enviados en la conexión. Además, el tamaño medio de mensaje de *GServer* es 3,43 veces mayor que la media de los 5 protocolos (que es de 160,62 bytes). Examinando 6.4.2. Experiencia empírica, se puede concluir que estos datos son la causa de los problemas relacionados con la experiencia del usuario.

Ya con el foco sobre MUSE-RP, se observa que sus cifras se asemejan a las de *Ruffles*, el protocolo RUDP ajeno con mejores resultados que se ha probado. Aunque es cierto que es el protocolo que más paquetes ha enviado, seguramente porque cada mensaje enviado debe ser reconocido (esto también se comprueba viendo que los números de paquetes enviados por los dos puntos son prácticamente idénticos). Sin embargo, esto no perjudica tanto a la cantidad de información enviada, ya que el tamaño medio de paquete es el más pequeño después de UDP. En un futuro se podría diseñar un método de fusión de ACKs que disminuyese el tráfico enviado por este protocolo. En el caso de, por ejemplo, TCP, se usan mecanismos de fusión de ACKs, o de ACKs retardados para este propósito.

En conclusión, los protocolos RUDP (exceptuando *GServer*) tienen estadísticas similares a lo que podría ser un híbrido entre TCP y UDP, aunque marcadas por las implicaciones de crear una capa extra sobre UDP, con el añadido de cabeceras y nuevas funcionalidades en capa de aplicación. En general *Ruffles* y MUSE-RP han cumplido con las expectativas, ofreciendo una buena experiencia al usuario en cuanto a *gamefeel*. De hecho, la prueba con TCP estuvo marcada por algunos cortes derivados de retrasmisiones, y la de UDP conllevó varios comportamientos erráticos por la pérdida de paquetes, por tanto, se reafirma que los protocolos RUDP pueden ser una buena alternativa a la hora de desarrollar videojuegos multijugador online.

7. Conclusiones y trabajo futuro

El objetivo principal de este trabajo, implementar un protocolo RUDP orientado a videojuegos (MUSE-RP), y compararlo con el rendimiento de otro tipo de protocolos se ha cumplido con éxito. De hecho, MUSE-RP ha resultado dar buenos resultados, similares al de otros protocolos RUDP (como es el caso de *Ruffles*), y llegando a tener potencial como para contemplarlo como alternativa a TCP o UDP en el desarrollo de videojuegos multijugador online.

La implementación se llevó a cabo en varias iteraciones (resumidas en 4. Implementación de MUSE-RP), las cuales repercutieron en gran medida al diseño del mismo para poder sortear las dificultades a las que este tipo de protocolos deben hacer frente. Este desarrollo resultó en un producto que cumple con las funcionalidades diseñadas. Sin embargo, destacar algunas funcionalidades o características que podrían ser parte de una futura actualización de MUSE-RP:

- Creación de métodos específicos que solucionen el problema que supone trabajar con hilos en *Unity*. Actualmente puede usarse el método *ProcessNextMessage()* de los *Host* en el bucle de juego, pero esto se podría mejorar.
- Uso de ACKs retardados que disminuya la cantidad de paquetes enviados en una sesión de juego. Esto podría conseguirse esperando a la llegada de nuevos mensajes en X milisegundos, y reconociendo varios a la vez.
- Herramientas que ayuden a los desarrolladores en la conversión de objetos a bytes. Con MUSE-RP solo pueden mandarse conjuntos de bytes, pero la manera de convertir lo que se quiere enviar en bytes es tarea del desarrollador.

Otro objetivo cumplido fue implementar un juego sencillo que pudiera servir de prueba para los protocolos estudiados. Este juego, además, pudo adaptarse perfectamente a todos los protocolos, y se diseñó de tal manera que la calidad del protocolo pudiese afectar a la experiencia del usuario. Como trabajo futuro, podría trabajarse más en este videojuego, añadiéndole mayor complejidad para poder probar las diferentes configuraciones de protocolos en un entorno más difícil. Además, quedó pendiente implementar algún tipo de mecanismo de capa de aplicación que solucione problemas de

latencia específicos de los videojuegos. Actualmente el juego no implementa este tipo de mecanismos, ya que es una tarea compleja que podría tratarse enteramente en otro trabajo. De estos mecanismos (como son *Dead Reckoning* o *SnapShot*) se habló en 1.2 Características de las redes en videojuegos.

Por último, las pruebas realizadas con los diferentes protocolos, usando *Wireshark,* arrojaron varios resultados interesantes sobre el desempeño de los diferentes protocolos en un juego dinámico. MUSE-RP y *Ruffles* resultaron ser una buena alternativa a los problemas que surgían con el uso de TCP y UDP, mientras que *GServer*, un protocolo RUDP orientado a videojuegos, resultó tener varios problemas que perjudicaban la experiencia de juego. Como posible trabajo futuro, sería interesante hacer más pruebas en otro tipo de entornos. En este trabajo se ha usado una arquitectura Cliente-Servidor, donde el servidor también era un jugador. Sería interesante realizar las mismas pruebas en otro tipo de arquitecturas, como la de un servidor centralizado que no fuese un cliente en sí mismo, si no que gestionase la partida (e incluso varias). El tráfico surgido de este tipo de arquitecturas es un entorno perfecto para poner a prueba los protocolos RUDP.

En conclusión, los protocolos RUDP han resultado ser una opción muy interesante a contemplar si se quiere hacer un juego multijugador online, sobre todo, si se quiere conseguir un híbrido entre TCP y UDP. La implementación de MUSE-RP, además, ha resultado ser un éxito, y aunque en un futuro pueda actualizarse y mejorar ciertos aspectos, en esta primera versión ya es un protocolo listo para su incorporación en un videojuego real.

8. Bibliografía

[1] Kurose, J. F., & Ross, K. W. (1986). Computer Networking. *A Top-Down*.

[2]https://www.gamesparks.com/blog/tips-for-writing-a-highly-scalable-server-authoritative-game-part-1/ [Consultado el 02-07-2022]

[3] Asensio, E., Ordua, J. M., & Morillo, P. (2008, September). Analyzing the network traffic requirements of multiplayer online games. In *2008 The Second International Conference on Advanced Engineering Computing and Applications in Sciences* (pp. 229-234). IEEE.

[4] de Albuquerque Torreão, V. Comparison between client-server, peer-to-peer and hybrid architectures for MMOGs.

[5] Swink, S. (2008). *Game feel: a game designer's guide to virtual sensation*. CRC press.

[6]Kim, J. R., Park, I. K., & Shim, K. H. (2007, Septiembre). The effects of network loads and latency in multiplayer online games. In *International Conference on Entertainment Computing* (pp. 427-432). Springer, Berlin, Heidelberg.

[7]Mount,Dave, & Eastman, Roger (2018). Multiplayer Games and Networking.

[8] Svoboda, P., Karner, W., & Rupp, M. (2007, June). Traffic analysis and modeling for World of Warcraft. In *2007 IEEE International Conference on Communications* (pp. 1612-1617). IEEE.

[9]https://gafferongames.com/post/snapshot_compression/[Consultado el 30-06-2022]

[10] https://www.gamedeveloper.com/programming/dead-reckoning-latency-hiding-for-networked-games[Consultado el 30-06-2022]

[11] Lee, G. (2014). *Cloud networking: Understanding cloud-based data center networks*. Morgan Kaufmann.

[12] Martín García, Celtia. (2022). Estudio y diseño de protocolos RUDP orientados a videojuegos.

[13] Reliable UDP Transport (RUDP) (tibco.com) [Consultado el 01-07-2022]

[14] Pack, S., Hong, E., Choi, Y., Kim, J. S., & Ko, D. (2002, December). Game transport protocol: a reliable lightweight transport protocol for massively multiplayer online games (MMPOGs). In *Multimedia Systems and Applications V* (Vol. 4861, pp. 83-94). SPIE.

[15] Huh, J. H. (2018). Reliable user datagram protocol as a solution to latencies in network games. Electronics, 7(11), 295

[16]Reliable UDP (RUDP): The Next Big Streaming Protocol? (streamingmediaglobal.com) [Consultado el 01-07-2022]

[17] GitHub - fexolm/GServer: Easy for use, game developing oriented reliable udp library[Consultado el 01-07-2022]

[18] GitHub - MidLevel/Ruffles: Lightweight and fully managed reliable UDP library. [Consultado el 01-07-2022]

[19] https://github.com/vis2k/Telepathy[Consultado el 01-07-2022]

[20] https://docs.microsoft.com/es-es/documentation/[Consultado el 01-07-2022]

[21]https://www.mooc.org/blog/best-programming-languages-for-game-development#:~:text=C%2B%2B%20is%20the%20most%20popular,of%20AI%2Dpowered%20game%20bots. [Consultado el 01-07-2022]

[22]https://www.incredibuild.com/blog/top-7-gaming-engines-you-should-consider-for-2020[Consultado el 01-07-2022]

[23] https://mirror-networking.com/[Consultado el 01-07-2022]

[24] https://www.photonengine.com/PUN[Consultado el 01-07-2022]

[25]https://ims.improbable.io/insights/kcp-a-new-low-latency-secure-network-stack[Consultado el 01-07-2022]

[26] https://www.wireshark.org/[Consultado el 01-07-2022]

[27] https://www.vpn.net/[Consultado el 02-07-2022]

[28]https://docs.microsoft.com/es-es/dotnet/api/system.net.networkinformation.ping?view=net-6.0[Consultado el 01-07-2022]

[29] Merola, Antonio.(2016) El ICMP, uso y abuso. *hakin9, 2.*

[30]https://docs.microsoft.com/es-es/dotnet/api/system.timers.timer?view=net-6.0[Consultado el 01-07-2022]

Protocolo MUSE-RP: https://github.com/Celtia-Martin/MUSE-RP-RUDP-protocol

Juego de prueba: https://github.com/Celtia-Martin/MUSERP-Test

9. Anexo: Código y salidas por pantalla de las pruebas con MUSE-RP

9.1. Código de las pruebas

9.1.1. Código común

```
public static void Main(string[] args)
{
            MessageHandler clientMessageHandler = new MessageHandler();

            HostOptions clientHostOptions = new HostOptions(2, 10000, 2000, 1, 0, 0, 10, 200, 50, clientMessageHandler);
            ConnectionInfo serverInfo = new ConnectionInfo("localhost", 49156, 49157);
            Client client = new Client(clientHostOptions, serverInfo, 3000, 3, true);
            Thread clientChatThread = new Thread(() => ChatClientThread(client));
            OnConnectedDelegate onConnect = () =>
             {
                 Console.WriteLine("OnConnected Iniciado, puede empezar a escribir cosas");
                 clientChatThread.Start();

             };
            OnConnectionFailure onFailure = () =>
             {
                 Console.WriteLine("OnConnected Iniciado, puede empezar a escribir cosas");
                 client.TryConnect();
             };
            client.AddOnConnectionFailureHandler(onFailure);
            clientMessageHandler.AddHandler(115, (m, c) => OnMessageReceivedClient(client, m, c));
            client.AddOnConnectedHandler(onConnect);
            client.Start();
            client.TryConnect();
            Console.WriteLine("Cliente comenzado");
}
```

Algoritmo 3:Código de creación de un cliente de las primeras pruebas de MUSE-RP

```
public static void Main(string[] args)
{
            MessageHandler serverMessageHandler = new MessageHandler();
            HostOptions serverHostOptions = new HostOptions(100, 10000, 2000, 1, 49156, 49157, 10, 200, 50, serverMessageHandler);
            Server server = new Server(serverHostOptions, true);

            serverMessageHandler.AddHandler(115, (m, c) =>

                   OnChatReceivedServer(server, m, c)

            );

            server.Start();
            Console.WriteLine("Server comenzado");
            Thread serverChatThread = new Thread(() => TestThread(server));
            serverChatThread.Start();
}
```

Algoritmo 4: Código de creación de un servidor de las primeras pruebas de MUSE-RP

9.1.2. Prueba de chat sencillo

```
public static void OnChatReceivedServer(Server server, MessageObject message, Connection source)
{
            string newMessage = source.ToString() + " ha dicho: " + ASCIIEncoding.ASCII.GetString(message.getData());
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(newMessage);
            Console.ForegroundColor = ConsoleColor.White;
            server.SendToAll(115, false, Encoding.ASCII.GetBytes(newMessage));

}
```

Algoritmo 5: Manejador de recepción de mensajes de chat por parte del servidor

```
public static void OnMessageReceivedClient(Client client, MessageObject message, Connection source)
{
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(ASCIIEncoding.ASCII.GetString(message.getData()));

}
```

Algoritmo 6: Manejador de recepción de mensajes de chat por parte del cliente

```
public static void ChatClientThread(Client client)
{
            string newMessage;
            while (true)
            {
                newMessage = Console.ReadLine();
                client.SendToServer(115, false, Encoding.ASCII.GetBytes(newMessage));
            }
}
```

Algoritmo 7: Hilo de envío de mensajes escritos por parte del cliente

```
public static void TestThread(Server server)
{
            string newMessage;
            newMessage = "Server ha dicho: " + Console.ReadLine();
            bool reliableTest = false;

            while (true)
            {

                server.SendToAll(115, reliableTest, ASCIIEncoding.ASCII.GetBytes(newMessage));

                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine(newMessage);
                Console.ForegroundColor = ConsoleColor.White;
                newMessage = "Server ha dicho: " + Console.ReadLine();

            }
}
```

Algoritmo 8: Hilo de envío de mensajes escritos por parte del servidor

9.1.3. Prueba de envío masivo de mensajes

```
public static void OnMessageReceivedClient(Client client, MessageObject message, Connection source)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            int data = BitConverter.ToInt32(message.getData());
            Console.WriteLine("Server dijo: " + data);
            Console.WriteLine(data);

            if (data != oldCont + 1)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine("MENSAJE PERDIDO " + oldCont + 1);
                Console.WriteLine("MENSAJE ACTUAL " + data);
                client.Stop();
            }
            oldCont = data;
            Console.ForegroundColor = ConsoleColor.White;

        }
```

Algoritmo 9: Manejador de recepción de mensajes masivos por parte del cliente

```
   public static void TestThread(Server server)
   {
            string newMessage;
            newMessage = "Server ha dicho: " + Console.ReadLine();
            int cont = 0;
            int testTime = 3;
            bool reliableTest = false;

            while (true)
            {

                server.SendToAll(115, reliableTest, BitConverter.GetBytes(cont));
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine(newMessage);
                Console.ForegroundColor = ConsoleColor.White;
                cont++;
                Thread.Sleep(testTime);

            }
   }
```

Algoritmo 10: Hilo de envío masivo de mensajes por parte del servidor

9.2. Salida por consola de las pruebas

9.2.1 Prueba de chat sencillo

```
Trying to connect
Cliente comenzado
Connected
Socket stopped because there was an exception receiving: Una operación de
CancelBlockingCall.
Channel started
Reliable channel connected
Sending ACK, Current ACK and LastRecognised 1414577457 2591051365
Channel started
Connected with both channels
ACK received: 4040229095
Success: 4292714512
Sending ACK, Current ACK and LastRecognised 4292714512 4040229095
OnConnected Iniciado, puede empezar a escribir cosas
Ping received: 34
Hello World
Ping received: 43
ACK received: 4040229096
Success: 4292714513
Sending ACK, Current ACK and LastRecognised 4292714513 4040229096
25.80.114.21062103 ha dicho: Hello World
Ping received: 47
Ping received: 46
Success: 4292714514
Sending ACK, Current ACK and LastRecognised 4292714514 4040229096
Server ha dicho: Hello
Sending ACK, Current ACK and LastRecognised 4292714514 4040229096
```

Ilustración 20: Salida por pantalla de la prueba de chat sencilla sobre MUSE-RP, con información sobre los mensajes de ping recibidos y el estado de la conexión

```
Server ha dicho: What's Going on
ACK received: 4292714515
Ping received: 45
Ping received: 94
Ping received: 30
Ping received: 75
Ping received: 45
On TimeOut
Connection with 25.80.114.210:52232 removed
Connection with 25.80.114.210:52232 removed
```

Ilustración 21: Salida por pantalla del servidor desconectándose de un cliente por timeout en MUSE-RP

```
Server comenzado
Received: 3987204835 at reliable channel
Channel started
Reliable channel connected 25.80.114.210:58826
Mandando mensaje 1118316252
Received: 1473852430 at not reliable channel
Ack: 1118316252
Procesando mensaje
Channel started
ACK received: 1118316252
Mandando mensaje 3758435059
Connected No Reliable channel: 25.80.114.21058026
```

Ilustración 22: Salida por pantalla de la creación y conexión de un cliente a un servidor MUSE-RP

9.2.2 Prueba de envío masivo de mensajes

```
Procesando mensaje
Success: 2646319507
Sending ACK, Current ACK and LastRecognised 2646319507 2923676538
Received: 2646319498 at not reliable channel
Procesando mensaje
Sending ACK, Current ACK and LastRecognised 2646319507 2923676538
Received: 2646319498 at not reliable channel
Procesando mensaje
Sending ACK, Current ACK and LastRecognised 2646319507 2923676538
Server dijo: 329
329
MENSAJE PERDIDO 328
MENSAJE ACTUAL 329
Desconnected
Server dijo: 330
330
Server dijo: 331
331
```

Ilustración 23: Salida por pantalla de un caso de pérdida de mensajes por el canal parcialmente fiable en MUSE-RP