Universidad Rey Juan Carlos

Estudio y diseño de protocolos RUDP orientados a videojuegos

MEMORIA DEL TRABAJO DE FIN DE GRADO EN INGENIERÍA DE COMPUTADORES

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 haberme ayudado a la hora de hacer trazas de paquetes para este trabajo.

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

El área de las redes de computadores desde la perspectiva del desarrollo de videojuegos es un campo muy rico en particularidades. A raíz de la naturaleza de dicha perspectiva surgen nuevos problemas que necesitan ser abordados para brindar productos de calidad.

La naturaleza interactiva de los videojuegos es lo que más influye en los requisitos del diseño de esta comunicación. Por una parte, requieren de alta velocidad y baja latencia, pero por otra, hay mucha información cuya pérdida no sería tolerable en un entorno interactivo.

Ambos requisitos, a primera vista, pueden parecer incompatibles, sobre todo si se tienen conocimientos sobre los protocolos utilizados en el transporte de información dentro de Internet: TCP y UDP.

TCP, por una parte, es un protocolo complejo que garantiza que toda la información enviada va a llegar a su destino en orden (es decir, es fiable), pero que sacrifica la velocidad de dicho transporte por culpa, por ejemplo, de las retrasmisiones de información.

UDP, sin embargo, es un protocolo ligero, sencillo y rápido, que no garantiza que la información enviada llegue a su destino, ni que lo haga en orden.

Actualmente hay abierto un debate sobre cuál de estos protocolos es el más adecuado para el desarrollo de videojuegos. Sin embargo, hay que añadir un tercer tipo de protocolo: los protocolos RUDP.

Los protocolos RUDP utilizan UDP, pero añadiendo funcionalidad extra para incorporar fiabilidad en el envío de información, y actualmente se conoce que están siendo utilizados en el área de videojuegos, ya que son un híbrido entre TCP y UDP.

En este documento se explorará este debate investigando, además, qué es lo que se está utilizando actualmente en los videojuegos. Posteriormente, se estudiarán los protocolos RUDP, lo cual será de ayuda para diseñar uno propio. Este cumplirá con los requisitos necesarios para adaptarse a las necesidades de los videojuegos, haciendo especial hincapié en su capacidad de configuración.

Contenido

1. Introducción 1
1.1. Introducción a redes de computadores 1
1.1.1 Redes de computadores 1
1.1.2. Modelo de capas 2
1.1.3. Uso de la red en videojuegos 3
1.1. Protocolos de transporte 4
1.2.1. UDP 4
1.2.2. TCP 5
1.2.3. Algoritmo de ventana deslizante 5
1.2.4. RUDP 7
1.2.5. Protocolos de transporte en videojuegos 9
2. Estado del arte 11
2.1. Casos de estudio 11
2.1.1. Comunicación con TCP en *Minecraft* 11
2.1.2. Comunicación con UDP en *Stardew Valley* 12
2.2. Protocolos RUDP estudiados 14
2.2.1. RUDP.NET 14
2.2.2. GServer 15
2.2.3. Ruffles 16
2.2.4. KCP 16
2.2.5. ENET 17
3. Motivación y objetivos 19
3.1. Encuesta a desarrolladores junior 19
3.2. Características deseadas 21
3.2.1. Dos canales de comunicación 22

3.2.2 Mecanismos orientados a conexión 23
3.2.3. *Handlers* específicos por tipo de mensaje 25
3.2.4. Comunicación entre capas 26
3.3. Metodología 27
3.3.1. Tecnologías usadas 27
3.3.2. Metodología de desarrollo 28
4. Diseño de MUSE-RP 31
4.1. Algoritmos de entrega fiable 31
4.1.1. Algoritmos Emisor 32
4.1.2. Algoritmo receptor entrega fiable 33
4.1.3. Algoritmo receptor entrega parcialmente fiable 35
4.2. Estructura de un paquete MUSE-RP 42
4.2.1. Estructura 42
4.2.2. *Flags* 43
4.2.3. Campos 43
4.3. Diseño del código 44
4.3.1 Descripción de clases 44
4.3.2 Diagramas UML 49
4.4. Pruebas y resultados 51
5. Conclusiones y trabajo futuro 55
6. Bibliografía 57
7. Anexo: Resultados de la encuesta a desarrolladores junior 59
7.1. Preguntas Generales 59
7.2. Preguntas específicas 61
7.3. Conclusiones de la encuesta 68
8. Anexo: Código en Python de prueba del algoritmo parcialmente fiable 69

Tabla de Ilustraciones

Ilustración 1: Estructura de las capas de internet tanto en modelo OSI como en TCP/IP[1] 3
Ilustración 2: Algunos paquetes enviados desde el servidor al cliente en una partida online de *Minecraft*. 12
Ilustración 3: Algunos paquetes enviados desde el cliente al servidor en una partida online de *Minecraft*. 12
Ilustración 4: Algunos de los paquetes UDP capturados en el flujo de una partida multijugador en *Stardew Valley* 13
Ilustración 5: Gráfica con los resultados a la primera pregunta de la encuesta a desarrolladores junior. 20
Ilustración 6:Gráfica con los resultados a la cuarta pregunta de la encuesta a desarrolladores junior. 21
Ilustración 7: Flujo de paquetes en un evento de inicio de conexión (*HandShake*) en MUSE-RP. 23
Ilustración 8: Comunicación entre las diferentes capas de red en MUSE-RP 26
Ilustración 9: Flujo de paquetes entre dos host mediante el algoritmo fiable de MUSE-RP, con un tamaño de ventana de 5 paquetes 35
Ilustración 10: Flujo de paquetes entre dos host con el algoritmo parcialmente fiable de MUSE-RP, donde el tamaño de ventana es de 3 y el porcentaje de fiabilidad 60% 39
Ilustración 11: Flujo de paquetes entre dos host con el algoritmo parcialmente fiable de MUSE-RP, donde el tamaño de ventana es de 5 y el porcentaje de fiabilidad 50% 40
Ilustración 12: Flujo de paquetes entre dos host con el algoritmo parcialmente fiable de MUSE-RP, donde el tamaño de ventana es de 5 y el porcentaje de fiabilidad 50% 41
Ilustración 13: Esquema de la estructura de un paquete MUSE-RP. 42
Ilustración 14: Diagrama UML de las clases relacionadas con los puntos terminales del protocolo MUSE-RP (Capa de aplicación) 49
Ilustración 15:Diagrama UML de las clases relacionadas con los canales de comunicación del protocolo MUSE-RP (Capa intermedia) 50
Ilustración 16: Salida por consola a un caso de prueba de comprobación de números de secuencia para el algoritmo parcialmente fiable. 52
Ilustración 17: Resultados sobre el género de los encuestados 59

Ilustración 18: Resultados sobre los perfiles profesionales de los encuestados 60
Ilustración 19: Resultados sobre la primera pregunta de la encuesta 61
Ilustración 20: Resultados sobre la segunda pregunta de la encuesta............................ 62
Ilustración 21: Resultados sobre la tercera pregunta de la encuesta.............................. 63
Ilustración 22: Resultados sobre la cuarta pregunta de la encuesta............................... 64
Ilustración 23: Resultados sobre la quinta pregunta de la encuesta............................... 65
Ilustración 24: Resultados sobre la sexta pregunta de la encuesta 66
Ilustración 25: Resultados sobre la séptima pregunta de la encuesta 67

Tabla de Algoritmos

Algoritmo 1: Algoritmo de envío de paquetes del emisor 32
Algoritmo 2: Algoritmo de procesado de paquetes ACK del emisor 33
Algoritmo 3: Procesado de paquetes recibidos por el receptor en el modo fiable de MUSE-RP 34
Algoritmo 4: Procesado de mensajes recibidos por el receptor en el modo parcialmente fiable de MUSE-RP 38
Algoritmo 5: Programa principal de la prueba en Python del algoritmo parcialmente fiable.. 69
Algoritmo 6: Código de recogida de datos de entrada de la prueba en Python del algoritmo parcialmente fiable. 69
Algoritmo 7: Algoritmo para coger paquetes del buffer de la prueba en Python del algoritmo parcialmente fiable. 69
Algoritmo 8: Algoritmo para añadir paquetes al buffer de la prueba en Python del algoritmo parcialmente fiable. 70
Algoritmo 9: Algoritmo que simula al receptor de la prueba en Python del algoritmo parcialmente fiable. 70
Algoritmo 10: Implementación del algoritmo parcialmente fiable de la prueba en Python del algoritmo parcialmente fiable. 71

1. Introducción

En este trabajo se va a explorar el mundo de las redes de computadores dentro de un área específica: los videojuegos. Estará centrado en investigar qué protocolos se están usando actualmente en este área, para luego hacer hincapié en un tipo de protocolo específico: los protocolos RUDP *(Reliable User Datagram Protocol).* Finalmente, se presentará una propuesta de diseño de un protocolo RUDP, configurable y orientado a videojuegos. Pero antes, ha de hacerse una pequeña introducción sobre las redes de computadores y su funcionamiento.

1.1. Introducción a redes de computadores

Es esencial comprender como funcionan las redes en computadores e Internet para poder abordar una investigación más profunda sobre los protocolos que utiliza.

1.1.1 Redes de computadores

Como se explica en "Computer Networking. *A Top-Down*."[1] Internet (la red de computadores más ambiciosa que existe) podría considerarse la mayor obra de la informática. Muchos dispositivos (ya no solo computadores) están conectados a ella. Pero, ¿cómo funciona? Estos dispositivos (llamados *hosts* o sistemas terminales) están conectados entre sí mediante enlaces de muchos tipos diferentes, y por medio de conmutadores de paquetes. Para que haya una comunicación entre estos enlaces ha de establecerse un canal de punto a punto, es decir, ha de haber un receptor y un emisor.

La información (que en esencia, son flujos de bytes) es enviada por el emisor a través de paquetes, fruto de la segmentación y de la adición de cabeceras de protocolo por cada segmento. Los conmutadores recogerán estos paquetes y los reenviarán para que lleguen al receptor. Los conmutadores más utilizados son los *routers* y los *switches*.

En Internet existen varios protocolos, cada uno ligado a una capa diferente (ver 1.1.2. Modelo de capas). Estos protocolos establecen una comunicación por medio de mensajes específicos. En "Computer Networking. *A Top-Down*."[1] se propone una analogía con la comunicación humana. Si una persona quiere pedir la hora a otra, primero ha de saludarse con un "Hola", a lo cual la otra persona responderá con otro "Hola", y

luego se llevará a cabo la pregunta, cuya respuesta será la información pedida. Ha de recalcarse que ambos deben hablar el mismo idioma, o no se entenderán.

1.1.2. Modelo de capas

El estándar dictamina organizar Internet en una serie de capas, cada una de ellas con sus funciones, responsabilidades y protocolos. El conjunto de estos protocolos se denomina pila de protocolos, diseñada de tal manera que las capas superiores se nutren de los servicios de las capas inferiores. Es el llamado modelo TCP/IP, y consta de 5 capas. Las capas, de bajo a alto nivel, son: capa física, capa de enlace, capa de red, capa de transporte y capa de aplicación. En algunos modelos, como el modelo OSI, se incorporan más capas por debajo de aplicación: presentación y sesión. Una representación de dichas capas puede verse en la Ilustración 1.

Para este trabajo, solo hará falta fijarse en dos capas: aplicación y transporte. Esto es debido a que los desarrolladores software solo pueden acceder a capa de aplicación y/o configurar la capa de transporte. Las demás capas son difíciles de manejar, e incluso pueden estar inaccesibles al estar a tan bajo nivel.

Es en la capa de aplicación donde se gestionan los mensajes en el contexto de la aplicación. Por ejemplo, se encuentran protocolos tales como HTTP *(Hypertext Transfer Protocol),* encargado de hacer peticiones web. Es aquí donde los videojuegos gestionarán la información recibida y enviada acorde a sus requisitos.

La capa de transporte, sin embargo, se encarga de transportar los paquetes entre los puntos terminales de una aplicación. Dos de los protocolos más importantes de esta capa son: TCP*(Transfer Control Protocol)* y UDP *(User Datagram Protocol).*

Para la comunicación entre estas dos capas (aplicación y transporte), existe una interfaz llamada *socket,* que suministra la información recogida por la capa de transporte a la capa de aplicación.

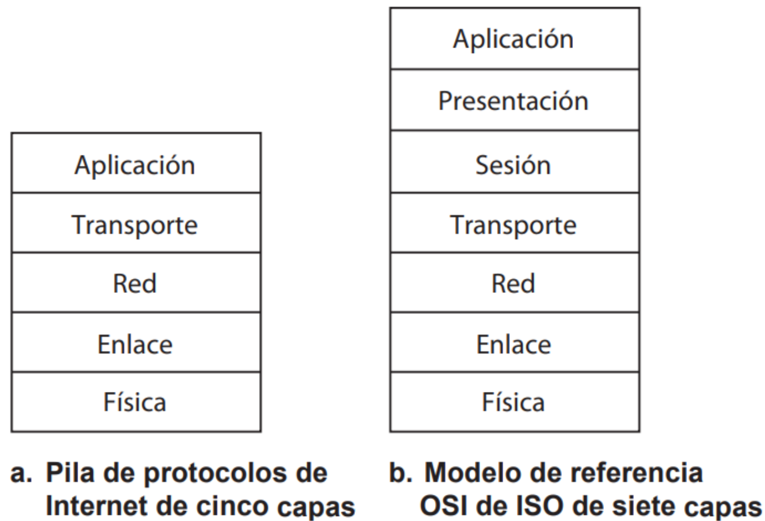

Ilustración 1: Estructura de las capas de internet tanto en modelo OSI como en TCP/IP[1]

1.1.3. Uso de la red en videojuegos

Actualmente muchos videojuegos exitosos son multijugador o cuentan con un modo online. La posibilidad de poder jugar con otras personas o con amigos atrae a muchos jugadores a este tipo de aplicaciones. Pero la implementación de esta comunicación no es trivial, e incluso plantea nuevos problemas que otro tipo de comunicaciones no presentan.

Las redes dentro del área de los videojuegos son un tema muy rico y con muchas particularidades derivadas de las características de estos mismos. Su naturaleza es única dentro de tráfico de red. Esto es debido a que es difícil compararlo con otros tipos de tráfico. En el tráfico multimedia, por ejemplo, se suele trasmitir video, imágenes y sonido. En cuanto al tráfico web, está enfocado a la interacción de los usuarios con las páginas web: gestión de formularios y descarga de datos, entre otros.[1] Sin embargo, en el tráfico de los videojuegos, lo que se va a trasmitir sobre todo son eventos e información de otros jugadores. Por ejemplo (y teniendo en mente una arquitectura cliente-servidor): en un FPS *(First Person Shooter)* es importante que el servidor mande a todos los jugadores información dinámica sobre el mapa (eventos meteorológicos, por ejemplo), la localización de otros jugadores y hacia donde se dirigen, qué aspecto tiene cada uno de esos jugadores, qué armas usan, qué disparos se producen y cuáles han sido certeros o no (traduciéndose en una disminución de la salud en el jugador correspondiente),entre otros.

En esencia: los mensajes que se mandan en un videojuego multijugador son de múltiples tipos, y es el propio videojuego el que decide como gestionar cada uno de ellos. Además, la tasa de refresco suele requerir ser lo más alta posible para poder brindarle al jugador un buen *Game Feel*[2]. Esto es debido en esencia a que los usuarios de un videojuego interactúan con él de forma activa, y si la red va lenta o tiene fallos, esta interacción puede ser incómoda o incluso nula. Sin embargo no hay un estándar estricto, hay muchos tipos de videojuegos, y sus requisitos pueden variar dependiendo de su diseño. Es por ello que en escenarios dinámicos no se tolera una alta latencia, pero sí pérdidas [3], aunque en otro tipo de juegos más estáticos (por ejemplo, juegos por turnos, como un ajedrez online), quizás la pérdida de paquetes sea menos tolerable, y la velocidad, algo más secundario.

1.1. Protocolos de transporte

En este apartado se explorarán los protocolos disponibles en la capa de transporte.

1.2.1. UDP

UDP es un protocolo simple y poco sofisticado, sin conexión, en el que el emisor pasará los mensajes directamente a capa de red, y en el que el receptor pasará a la aplicación todo paquete que le llegue de la capa de transporte. Es decir, hace su mayor esfuerzo para entregar los paquetes, pero no garantiza que hayan llegado, ni que hayan llegado en orden. Entre sus ventajas, se encuentra un mayor control en cuanto a qué y cuando se envían los datos, y una cabecera ligera. Además, al no haber control de congestión o establecimiento de conexión (entre otras cosas), suele tener poca latencia. Se usa, por ejemplo, para flujos multimedia. Está definido en el documento [RFC 768].[1]

1.2.2. TCP

UDP ofrece una comunicación sencilla, pero a cambio no ofrece ninguna garantía de que los paquetes que se entreguen o que lo hagan ordenadamente. Es decir, no es fiable. Para que el canal de comunicación establecido en capa de transporte sea fiable, el receptor de los paquetes ha de implicarse en la comunicación, respondiendo con mensajes de confirmación de entrega al emisor y descartando los paquetes desordenados. TCP implementa este tipo de mecanismos.

TCP está orientado a conexión, es fiable, y cuenta con un control de congestión. Es un protocolo complejo, y que cuenta con un montón de características interesantes, imposibles de abarcar en este documento. Sin embargo, estas mismas características pueden causar retrasos, por ejemplo, por la retrasmisión de paquetes perdidos, que conlleva a un mayor RTT (*Round-Trip Time*, es decir, tiempo de ida y vuelta de un paquete entre los dos puntos establecidos en la comunicación). Esto no sería un problema, de no ser porque todas estas características no pueden deshabilitarse. Es decir, su grado de configuración es muy limitado.

Para la entrega y ordenación garantizados usa un algoritmo de ventana deslizante, además del uso de un tipo de mensajes especiales llamados ACK (de *acknowledgement*), que serán los que envíe el receptor para confirmarle al emisor que su paquete ha llegado con éxito a su destino.[1]

En este trabajo se van a diseñar mecanismos similares a los de TCP para garantizar fiabilidad en la entrega y en la ordenación. El mecanismo en el que se ha centrado más este trabajo es el ya mencionado algoritmo de ventana deslizante, por tanto no se puede avanzar sin explicar de qué se trata.

1.2.3. Algoritmo de ventana deslizante

Hay varias propuestas de algoritmos de entrega y ordenación fiable, tanto teóricas como prácticas. Un vistazo a estos algoritmos, de menor a mayor complejidad, puede verse en el libro de “Computer Networking. *A Top-Down*.”[1]. Estos algoritmos tienen en común que manejan cómo deben comportarse los receptores y los emisores ante la emisión-recepción de paquetes, y que hacen uso de una “ventana deslizante” . Esta ventana de mensajes abarca desde el último paquete enviado no reconocido (llamado base), hasta el tamaño de ventana. El último paquete enviado no reconocido no puede

estar fuera de dicha ventana. El tamaño de la ventana es importante, ya que el emisor no podrá enviar más paquetes si esta está completa. A medida que se van reconociendo los paquetes, esta ventana se va "deslizando", siempre y cuando en estos nuevos paquetes reconocidos se encuentre la base.

Con este concepto ya explicado, se puede pasar a resumir algunas características de algunos de estos algoritmos: GBN, Repetición Selectiva y el algoritmo de ventana de TCP.

1.2.3.1. GBN

El receptor de GBN (*Go Back N*), al llegarle un paquete ordenado (es decir, su número de secuencia es el del último paquete entregado a capa superior más una unidad) lo entrega a capa superior y envía un ACK con su secuencia. En cualquier otro caso, descarta el paquete y reenvía el último ACK recibido. Además hace uso de un mecanismo llamado reconocimiento acumulativo. Si al emisor le llega un ACK 8 y el último ACK que le llego es 2, reconocerá los paquetes del 3 al 8, ya que se asegura que todos han llegado a destino por la propia naturaleza del algoritmo. Por último, ante un evento de fin de temporalización, el emisor reenviará todos los paquetes desde el paquete base al último enviado, lo cual podría ser contraproducente en caso de que solo se haya perdido uno, porque puede saturar la red.

1.2.3.2 Repetición Selectiva

En la repetición selectiva(o SR) existe un temporizador por paquete y el reconocimiento es selectivo. Es decir, si termina el temporizador del paquete 5, solo se reenviará el paquete 5, y si llega un paquete ACK reconociendo el paquete 5, solo se reconocerá este, y si no es la base de la ventana, no se "desliza". Eso quiere decir que el receptor reconocerá los paquetes, incluso si han llegado desordenados. De hecho, aunque los reciba y los reconozca, no los pasará a capa superior hasta que no se eliminen los posibles huecos entre estos paquetes y la base.

1.2.3.3 Algoritmo de ventana de TCP

Finalmente, TCP usa su propio algoritmo de ventana deslizante, incorporando mejoras a los dos algoritmos anteriores. Primero, en vez de números de secuencia consecutivos, TCP genera un *stream* de bytes, por lo que el número de secuencia se corresponde al último byte enviado, mientras que el número de ACK corresponde al siguiente byte esperado por el receptor. Es decir, si se manda un paquete de 17 bytes, el número de ACK de respuesta será 18, y si luego se envía un paquete de 20 el número de secuencia sería 37, y el ACK respuesta, 38. A la hora de reconocer paquetes funciona como GBN, ya que no reconocerá paquetes desordenados, pero sin descartarlos, si no que los irá guardando en un buffer para cuando lleguen los que falten. Además, para evitar la saturación del canal, el final de temporización irá ligado únicamente al paquete base, y solo se retrasmitirá este. Aparte de estas especificaciones, TCP cuenta con muchas funcionalidades extra, entre ellas la retrasmisión rápida, que consiste en retrasmitir un paquete si al emisor le llegan 3 ACK duplicados (es decir, 4 iguales).

Para una explicación más visual, visitar los applets que ofrece "Computer Networking. *A Top-Down*."[1] :

- Algoritmo GBN : https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/go-back-n-protocol/index.html
- Algoritmo SR: https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/selective-repeat-protocol/index.html
- TCP: https://media.pearsoncmg.com/aw/ecs_kurose_compnetwork_7/cw/content/interactiveanimations/flow-control/index.html

1.2.4. RUDP

A veces es necesaria una fina capa de fiabilidad sobre UDP (es decir, una capa extra entre las capas de transporte y aplicación en el modelo IP/TCP). Esta fina capa puede brindarle al desarrollador más flexibilidad en cuanto a qué configuraciones considera más oportunas de implementar en cuanto a fiabilidad, además de poder optar por un protocolo con menos latencia sin necesidad de renunciar a la fiabilidad.

A estos protocolos se les llama RUDP *(Reliable User Datagram Protocol),* y existen diversas implementaciones, sobre todo en el mundo de los videojuegos. Muchas empresas tienen protocolos propietarios[4], aunque, como se verá más adelante, también hay protocolos que cumplen funciones similares de código abierto, accesibles a través de GitHub.

¿Por qué hacer los protocolos sobre UDP y no sobre TCP? TCP es muy completo y complejo, UDP, por otro lado, es muy básico y apenas tiene control sobre el tráfico. Es indiscutible que UDP es una muy buena base para implementar una capa extra por esta simpleza.

Es aquí cuando se entra en el mundo de los RUDP. Un RUDP es un protocolo que crea una capa por encima de UDP para hacerlo fiable[5]. Podría decirse que son extensiones de UDP, protocolos de aplicación, o incluso protocolos de transporte. No parece haber un consenso en cuanto a cómo clasificarlos, ya que a lo largo de la realización de este trabajo se han encontrado protocolos que se autoproclaman protocolos de transporte (como es el caso de KCP[6]), estudios que aseguran que los RUDP corresponden a la capa de aplicación o a una capa separada[7] o, directamente, RUDPs que se proclaman extensiones/librerías de UDP (Ruffles, [8]). Sin embargo, la esencia es la misma: crear un mecanismo que añada fiabilidad a UDP para poder aprovechar su velocidad sin tener que sacrificar paquetes que puedan ser importantes. El cómo implementar este mecanismo es tarea del desarrollador, y su naturaleza puede variar mucho entre unos y otros.

Ya entrando en el terreno de los videojuegos, algunos aseguran que es la solución para mejorar la latencia en el tráfico multijugador [9][4], sin embargo, aún no hay muchas investigaciones que aporten profundidad al tema, por tanto se puede decir que es un campo que aún está en fase de crecimiento y evolución, y por eso no se ha establecido un estándar para su implementación [4].

Actualmente hay muchos tipos de RUDP, tanto libres (ENET, por ejemplo)[10] como comerciales (como es el caso de *Aspera*) [11]. De hecho, se asegura que actualmente están siendo muy utilizados en el mundo de los videojuegos, por ejemplo, en Corea [9]. Al no haber un estándar, no es extraño pensar que tanto las grandes como pequeñas empresas de videojuegos cuenten con su propio protocolo RUDP.

Pero ¿cómo se puede crear un protocolo RUDP? Pues una propuesta sencilla podría ser implementar los mecanismos que tiene TCP para la entrega fiable pero a nivel de aplicación. También podrían utilizarse algoritmos de ventana tales como GBN o

Repetición Selectiva, si fuesen más acertados. La ventaja de implementar esto en capa de aplicación, es que podemos configurar este control más cómodamente, a un alto nivel. A la misma conclusión llega Abhilash Thammadi en su propuesta de protocolo RUDP [12]. Sin embargo, tampoco se deben utilizar todos los mecanismos que incorpora TCP, porque entonces se podría estar implementando una réplica de este [7].

1.2.5. Protocolos de transporte en videojuegos

¿Qué tipo de protocolo se usa en videojuegos actualmente? Se usan tanto TCP como UDP, dependiendo de lo que se quiera conseguir: una comunicación orientada a conexión, repleto de características y funcionalidades cuya configuración es altamente limitada; o una comunicación simple y rápida, aunque con pérdidas y sin garantía de entrega ordenada. Del que más se habla en este campo es de UDP, dado a la excesiva complejidad y poca configurabilidad de TCP, además de su alta latencia.

Al final, la decisión de qué protocolo utilizar la toman los desarrolladores teniendo en cuenta la naturaleza de la aplicación que quieran implementar. Por ejemplo: si se desea desarrollar un juego por turnos (como podría ser el ajedrez o un juego de cartas) quizás la mejor opción sea utilizar TCP. En este tipo de juegos la velocidad no es muy importante, pero sí que se requiere que la entrega de los distintos eventos esté garantizada, porque si no se daría lugar a diversos errores. En el ejemplo del ajedrez, si un usuario se ha comido una pieza y este evento no llega correctamente al otro jugador, este podría jugar con dicha pieza, dando lugar a un error en el flujo del juego.

Véase otro caso diferente. Si se quiere desarrollar un FPS *(First Person Shooter),* hay que tener en cuenta que va a ser un juego muy dinámico, y los jugadores van a estar recibiendo mucha información por pantalla en muy poco tiempo. En este caso, algunos paquetes como la posición de los demás jugadores en todo momento, pueden permitirse pérdidas, siempre y cuando no entorpezcan el flujo del juego. Aquí podría llegarse a la conclusión de que UDP es una buena opción para el desarrollo del juego, sobre todo por su baja latencia. Sin embargo, hay otro tipo de mensajes cuya pérdida podría causar errores. Por ejemplo, si lo que se quiere mandar es un paquete que indica que el jugador ha muerto, su pérdida podría desembocar en un fallo fatal para el flujo del juego y para la experiencia del usuario. Esto podría solucionarse utilizando un protocolo RUDP, que aporta fiabilidad sin renunciar a la velocidad.

Véanse ahora casos reales. *World Of Warcraft*, así como otros MMORPG(Juegos de rol multijugador masivos) usan TCP. Esto se debe a qué se prioriza que la entrada de comandos del jugador llegue correctamente, y a que no se quieren propagar errores en largas sesiones de juego[2]. Otro caso de juego actual y conocido que utiliza TCP es *Minecraft* (ver 2.1.1. Comunicación con TCP en *Minecraft*).

Por otra parte, muchos otros juegos utilizan UDP. Es el caso de *Stardew Valley*, (ver 2.1.2. Comunicación con UDP en *Stardew Valley*) o de *League of Legends.*

2. Estado del arte

Después de haber analizado desde un plano más general cómo es la comunicación en los videojuegos multijugador, es necesario realizar un estudio de la situación actual de dichas comunicaciones en casos reales. Es por ello que en ese apartado se estudiarán trazas de dos juegos que usan protocolos de transporte diferentes, y se explorarán diferentes protocolos RUDP.

2.1. Casos de estudio

En este apartado se van a investigar en más profundidad dos casos ya mencionados en el apartado anterior : *Minecraft* y *Stardew Valley*. Para ello se ha utilizado el software de *WireShark* [13], el cual captura y analiza los paquetes recibidos por un sistema terminal.

2.1.1. Comunicación con TCP en *Minecraft*

Para poder jugar *Minecraft online*, se tienen dos opciones: o unirse a un servidor ya creado por otro usuario (introduciendo su dirección IP, y si es necesario, su puerto), o crear un servidor. Para ello, las páginas oficiales de *Minecraft* permiten descargar un archivo *.jar* que se encargará de todo (ya que *Minecraft* está escrito en java). Para este estudio se usó *Hamachi* para crear una red virtual. Todos con acceso a dicha red, y con la información correspondiente, podrán acceder al servidor. Se realizaron capturas tanto del lado del servidor como del cliente.

No es sorprendente ver como los paquetes TCP enviados son de un tamaño casi despreciable (observar la Ilustración 2), esto ya se podía adivinar, ya que en el caso del *World of Warcraft* ocurre lo mismo. Es una manera de reducir el RTT usando TCP.[2]

Averiguamos algo más, y es que la cantidad de mensajes sustanciales (no solo ACKs) enviados por el cliente (ver la Ilustración 3) es muy pequeña en comparación a la información mandada por el servidor (Ilustración 2). Esto puede ser debido a que Minecraft use un esquema de servidor autoritativo. En este esquema, el servidor es el encargado de calcular e informar de todos los eventos del mundo virtual, y los clientes se limitan a enviar las entradas de los jugadores (por ejemplo, qué elementos de la interfaz

han activado o qué teclas han pulsado). Este esquema suele utilizarse, por ejemplo, para evitar trucos en los juegos online.[14]

31	0.031867	25.70.15.166	25.70.17.96	TCP	68	25565 → 51787 [PSH, ACK] Seq=207 Ack=1 Win=64193 Len=14
32	0.031871	25.70.15.166	25.70.17.96	TCP	60	25565 → 51787 [PSH, ACK] Seq=221 Ack=1 Win=64193 Len=6
34	0.031946	25.70.15.166	25.70.17.96	TCP	60	25565 → 51787 [PSH, ACK] Seq=227 Ack=1 Win=64193 Len=6
35	0.031971	25.70.15.166	25.70.17.96	TCP	66	25565 → 51787 [PSH, ACK] Seq=233 Ack=1 Win=64193 Len=12
37	0.032021	25.70.15.166	25.70.17.96	TCP	60	25565 → 51787 [PSH, ACK] Seq=245 Ack=1 Win=64193 Len=6
38	0.032026	25.70.15.166	25.70.17.96	TCP	66	25565 → 51787 [PSH, ACK] Seq=251 Ack=1 Win=64193 Len=12
40	0.032090	25.70.15.166	25.70.17.96	TCP	60	25565 → 51787 [PSH, ACK] Seq=263 Ack=1 Win=64193 Len=6
41	0.032121	25.70.15.166	25.70.17.96	TCP	60	25565 → 51787 [PSH, ACK] Seq=269 Ack=1 Win=64193 Len=6

Ilustración 2: Algunos paquetes enviados desde el servidor al cliente en una partida online de *Minecraft*.

33	0.031932	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=227 Win=64992 Len=0
36	0.031994	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=245 Win=64974 Len=0
39	0.032031	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=263 Win=64956 Len=0
42	0.032128	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=275 Win=64944 Len=0
45	0.032169	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=291 Win=65472 Len=0
48	0.032859	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=316 Win=65447 Len=0
51	0.035901	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=334 Win=65429 Len=0
55	0.035918	25.70.17.96	25.70.15.166	TCP	54	51787 → 25565 [ACK] Seq=1 Ack=369 Win=65394 Len=0

Ilustración 3: Algunos paquetes enviados desde el cliente al servidor en una partida online de *Minecraft*.

2.1.2. Comunicación con UDP en *Stardew Valley*

El modo multijugador *de Stardew Valley* funciona de la siguiente manera: un usuario crea una granja y activa la opción de hacer de *Host* y poder invitar a sus amigos a la partida. Para unirse a la partida, los demás jugadores pueden conectarse tanto localmente (si se encuentran en la misma red) como en remoto (con una red diferente). Para el primer caso, será necesario ingresar la IP privada del Host, mientras que para el segundo, se puede ingresar con un código o una invitación. A primera vista, se puede teorizar que el usuario con la granja funcionará como un servidor, mientras que los que se conecten serán clientes. Para la gestión de invitaciones y códigos, es probable que exista un servidor del propio juego que ayude en esta conexión, sin embargo, no se puede saber hasta qué punto influye este servidor en la partida, porque es muy probable que una vez conectados los jugadores hagan uso de un canal que se haya creado entre ellos. Sabiendo esto, se analizará una captura realizada jugando a dicho juego con un jugador Host y un jugador cliente en local.

Como se puede observar en la Ilustración 4, en la comunicación entre el jugador que posee la granja y el jugador invitado se mandan paquetes UDP. No se sabe con certeza si el juego usa UDP en su estado más puro o si hace uso de algún protocolo RUDP.

55	1.869066	192.168.43.64	192.168.43.164	UDP	121	24642 → 52870 Len=79
56	1.872929	192.168.43.164	192.168.43.64	UDP	50	52870 → 24642 Len=8
57	2.271809	192.168.43.164	192.168.43.64	UDP	114	52870 → 24642 Len=72
58	2.272829	192.168.43.64	192.168.43.164	UDP	1240	24642 → 52870 Len=1198
59	2.272829	192.168.43.64	192.168.43.164	UDP	153	24642 → 52870 Len=111
60	2.276785	192.168.43.164	192.168.43.64	UDP	56	52870 → 24642 Len=14
61	2.278171	192.168.43.64	192.168.43.164	UDP	50	24642 → 52870 Len=8
62	2.317373	192.168.43.64	192.168.43.164	UDP	110	24642 → 52870 Len=68
63	2.320736	192.168.43.164	192.168.43.64	UDP	50	52870 → 24642 Len=8
64	2.322656	192.168.43.164	192.168.43.64	UDP	115	52870 → 24642 Len=73
65	2.327450	192.168.43.64	192.168.43.164	UDP	50	24642 → 52870 Len=8
66	2.835203	192.168.43.164	192.168.43.64	UDP	114	52870 → 24642 Len=72
67	2.840210	192.168.43.64	192.168.43.164	UDP	50	24642 → 52870 Len=8

Ilustración 4: Algunos de los paquetes UDP capturados en el flujo de una partida multijugador en *Stardew Valley*.

Sin embargo, se puede tener la sospecha de que, en efecto, usa un protocolo RUDP por lo siguiente: los paquetes suelen variar en tamaño, pero hay muchos mensajes con un tamaño fijo y pequeño (específicamente, 8 bytes), que pareciera que responden siempre a un mensaje de mayor tamaño (véase en la Ilustración 4 la pareja de paquetes 66 y 67, o 55 y 56). Como se vio en 1.2.4. RUDP, una forma de crear una capa de fiabilidad por encima de UDP es usando mecanismos propios de TCP, como el uso de ACK. Es decir: es muy probable que estos paquetes sean un tipo de ACK propio del protocolo RUDP que se esté usando.

Para reforzar la hipótesis, se realizó un análisis más exhaustivo de estos mensajes, y se comprobó que los datos que se envían en un flujo de información en este tipo de mensajes, van incrementándose uniformemente. Para ser exactos, si se transformaran los datos enviados a un decimal, el incremento sería de 256 unidades(100 en un sistema hexadecimal). Este incremento no parece aleatorio, por tanto refuerza la teoría de que nos encontramos ante un ejemplo de protocolo RUDP en un videojuego online.

Otra cosa en la que destaca esta captura es que el tamaño medio de los mensajes suele ser pequeño(no más de, aproximadamente, 100 bytes). Esto puede ayudar a que la comunicación sea más rápida, y por tanto, más fluida, como ya se especificaba en los múltiples estudios consultados [15]. Sin embargo, a veces aparecen paquetes más grandes (ver paquete 58 en la Ilustración 4), aunque ninguno supera los 1200 bytes.

Lo que contienen los paquetes con carga útil no es posible concretarlo, ya que su decodificación tendrá lugar dentro de la aplicación, que no es de código abierto. Sin embargo, se puede adivinar que cada tipo de mensaje corresponderá a un evento diferente. Por ejemplo, que el jugador se ha movido a X posición y ha plantado unas semillas.

2.2. Protocolos RUDP estudiados

Como ya se adelantó en el apartado 1.2.4. RUDP, no hay un estándar preestablecido de protocolo RUDP. Quizás sea por ello que se pueden encontrar varias implementaciones, cada una con sus pros y sus contras, e incluso especificidad por un área concreta, como son los videojuegos. A continuación se presentarán algunos de los protocolos RUDP estudiados.

2.2.1. RUDP.NET

Escrito en C#. El repositorio de este protocolo[16] no cuenta con un *README* explicativo (solo especifica que es una solución de protocolo RUDP escrito en C#). Por tanto las conclusiones que se pueden sacar son examinando el código.

Usa una arquitectura Cliente-Servidor, tiene un script para el comportamiento del servidor y otro para el cliente (*RudpServer* y *RudpClient*). El código es muy legible, y puede entenderse muy bien que hace cada línea. Puede observarse que usa un atributo del paquete para determinar si es fiable o no, y además usa una variable numérica para determinar el tipo del paquete. Con esta información, llama al *handler* que le corresponde para procesar el mensaje (en el apartado 3.2.3. *Handlers* específicos por tipo de mensaje podrá observarse que se implementa un método similar en el diseño de MUSE-RP). También hay un método para retrasmisión de mensajes. Lo hace comprobando todos los mensajes que están en una cola de mensajes fiables enviados, y si ya ha pasado demasiado tiempo desde que fueron enviados, lo reenvía. Estamos por tanto, ante un tipo de protocolo fiable selectivo (como la repetición selectiva).

Como posibles propuestas de mejora, sería adecuado hacer que *RudpServer* y *RudpClient* heredaran de una clase común para no repetir código, como está ocurriendo. Además, la comprobación selectiva de los mensajes que ya han sido enviados puede ser más costoso que tener un *timer* para una versión de ACKs acumulativos, sin embargo esta ya es una decisión de diseño más subjetiva y orientada a cómo se quiere que el protocolo se comporte.

2.2.2. GServer

Escrito en C#. El README de este repositorio[17] es bastante completo. Se explica que GServer es una librería diseñada específicamente para videojuegos, que ofrece un protocolo RUDP, y una estructura flexible y fácil de usar. Ofrece varios modos de fiabilidad en el envío de mensajes: sin modo, solo fiable, solo ordenado y fiable y ordenado. Es decir, se podrá garantizar la entrega fiable y la entrega en orden de forma flexible.

A diferencia de RUDP.NET, GServer usa la clase *Host*, y no diferencia entre Cliente y Servidor, dando flexibilidad al programador. Sus mensajes se estructuran de la siguiente manera: un *short* que indica el tipo de mensaje (el cual le redireccionará al *handler* específico, al igual que RUDP.NET), un objeto *Mode* que indica el modo de fiabilidad, y un array de bytes con la carga útil. También cuenta con la clase *DataStorage*, que ayuda con la serialización de la carga útil.

De un examen preliminar del código se pueden averiguar otras características. Por ejemplo, los paquetes cuentan con prioridad, y este será el criterio para ordenar los mensajes que llegan y ejecutarlos siguiendo esa ordenación. La clase *Connection* es la que se encarga de todo el manejo de los paquetes y de que la comunicación sea fiable (en los modos que se hayan decidido para cada paquete). Por lo que parece, también usa ACKs selectivos, ya que lo que ocurre al llegar un ACK es que el mensaje reconocido se elimina del buffer. De hecho, para el identificador de mensaje se usan dos parámetros: el id en sí y el *short* que diferenciaba entre tipos de mensajes.

2.2.3. Ruffles

Escrito en C#. En el repositorio de este protocolo [8] creado por el grupo *MidLevel* puede observarse, primero, que tiene una gran comunidad detrás: muchos usuarios lo han usado en proyectos propios, tiene 170 estrellas, e incluso su propio servidor de *Discord*. Es el protocolo de código abierto más completo que se ha encontrado en esta investigación.

El *README* es muy detallado. Comienza explicando la motivación de crear una librería RUDP. Menciona otras librerías ya existentes, tales como *Lidgren* o ENET. Su propósito es crear una librería comparable con estas, pero más ligera y sin funcionalidades innecesarias.

Cuenta con muchas características (explicadas cuidadosamente en el README), tales como: manejo de conexión, uso de Ipv4 y Ipv6, fusión de ACKs, estadísticas de conexión, fragmentación, o fusión de mensajes pequeño.

Al igual que el GServer, cuenta con varios modos de envío: fiable, fiable y ordenado, fiable ordenado y fragmentado, no fiable, no fiable ordenado y muchos más. Además, asegura que es seguro usarlo en *Unity*.

Analizando un poco el código se puede llegar a la conclusión de que es un protocolo con mucho trabajo detrás, y que supera con creces lo visto con anterioridad. Su primera versión salió a la luz en Mayo de 2019, mientras que la última (v11.1.5) salió en diciembre de 2020. Es una solución profesional y de código abierto, preparada para ser usada sin problemas.

Al igual que GServer, no se centra en una arquitectura cliente-servidor, si no en una clase llamada *RuffleSocket*, que es la que manejará toda la comunicación.

2.2.4. KCP

El de KCP es un caso especial, ya que como indica la documentación [18], no se encarga del envío-recepción de mensajes, solo del algoritmo de fiabilidad y *callbacks* que deberán ser llamadas cuando se precise. Es decir, en realidad es independiente del protocolo de la capa inferior, aunque normalmente se usará con UDP. También es independiente de la arquitectura que se vaya a usar. Se ha diseñado de tal forma que sea flexible y se pueda acomodar a muchos tipos de tecnología. Sin embargo, esto le puede delegar al programador más trabajo para poder utilizarlo para sus proyectos.

Su propósito es reducir la latencia y los retrasos de TCP, a cambio de aumentar un poco el ancho de banda. Está escrito en C, y usa tan solo dos archivos: *ikcp.h* y *ikcp.c*. En su *README* se enumeran algunas de sus características más importantes: retrasmisión rápida, elección de tener ACKs retardados o no, la posibilidad de modificar el modo de control de flujo para una trasmisión más veloz, entre otras.

En el propio *README* hay un enlace a proyectos en los que se ha implementado de forma satisfactoria. De hecho, se ha usado para la implementación, por ejemplo, de *Mirror Networking* [19], una librería de *Networking* que se puede obtener en la *asset store* de *Unity* gratuitamente. Por tanto, se trata de un protocolo con un acabado profesional, de libre acceso y con buenos resultados (como puede verse en el apartado *Protocol Comparision* del *README*).

2.2.5. ENET

ENET[10] ofrece una solución de RUDP de código abierto. Se trata de una capa simple y robusta por encima de UDP, que le brinda fiabilidad a varios niveles: tanto de entrega como de orden, ambos opcionales. Sin embargo, ignora todas aquellas funcionalidades que son propias de la capa de aplicación, como, por ejemplo, el cifrado de información.

Surgió del desarrollo de un juego FPS multijugador llamado *Cube*, y de la necesidad de una alternativa a TCP para poder tener baja latencia, a la vez que fiabilidad y manejo de conexión. Al no poder renunciar a estas características de TCP, se creó un protocolo específico para ello. Es por ello que se le considera como una mezcla entre UDP y TCP[10].

En cuanto a sus características más destacables: ENET usa varios *streams* de paquetes secuenciales en vez de uno solo, todos ellos con su número de secuencia; cuenta con varios canales de comunicación; y ofrece mecanismos para fragmentar mensajes.

ENET usa C como lenguaje de programación, y al fijarse en el repositorio de GitHub se puede observar que el código lleva manteniéndose desde su creación (hace más de una década) hasta el día de hoy (última modificación a principios de 2022 a fecha de hoy). De esto puede deducirse que su desarrollo sigue activo y que se sigue manteniendo.

3. Motivación y objetivos

Antes del diseño del protocolo, ha de establecerse un objetivo principal: crear un protocolo RUDP configurable y orientado a videojuegos. A partir de este objetivo principal, han de establecerse otros objetivos concretos sobre qué tipo de funcionalidades se quieren conseguir. Para ello, se hizo una investigación previa sobre las preferencias de los usuarios.

Este protocolo, además, tomará el nombre de MUSE-RP (de las siglas, *Multiplayer UDP Service Extension-Reliable Protocol*), que será el nombre que se utilizará para identificarlo en este documento.

3.1. Encuesta a desarrolladores junior

Para conocer las necesidades de los desarrolladores, se hizo una encuesta a estudiantes y exestudiantes de Diseño y Desarrollo de Videojuegos de la Universidad Rey Juan Carlos. Se quería descubrir, sobre todo, en qué medida querían tener control sobre el protocolo, y qué priorizarían al diseñar un videojuego online.

Se expuso un caso hipotético, y a continuación , una serie de preguntas:

> *Antes de contestar a las siguientes preguntas, me gustaría que te pusieses en la siguiente situación hipotética:*
>
> *Estas desarrollando un juego multijugador online y tienes que tomar varias decisiones:*
>
> *-Cómo se van a gestionar los mensajes y el tráfico en la red*
>
> *-Qué tipo de información se va a enviar*
>
> *-Que protocolo vas a usar para este cometido*
>
> *1. ¿En qué medida te gustaría tener el control de cómo se va a gestionar el tráfico de mensajes de tu juego?*
>
> *2. ¿En qué medida ves útil poder consultar las estadísticas del tráfico de tu juego?*
>
> *3. ¿Utilizarías una arquitectura Cliente-Servidor o Peer to Peer?*
>
> *4. ¿Qué primarías, que se garantizase que la información enviada ha sido recibida con éxito (como en TCP), o que se enviase lo más rápido posible, aún con pérdidas (como en UDP)?*

5. ¿En qué medida te gustaría poder personalizar la estructura de los mensajes que se fueran a utilizar en el intercambio de información?

6. ¿En qué medida ves útil que se puedan utilizar varios modos de envío de información dependiendo del tipo de mensaje? Por ejemplo: Un modo fiable con entrega garantizada en todos los casos, un modo poco fiable, con entrega garantizada para un porcentaje específico de mensajes, un modo sin entrega garantizada...

7. ¿Crees que deberían marcarse los mensajes que deben llegar sí o sí a su destino?

8. (Opcional) ¿Hay alguna característica específica que quisieras añadir al protocolo que se fuera a utilizar para controlar el tráfico online?

A continuación se muestran los resultados de las preguntas más destacables de la encuesta (para ver los resultados completos, ir al 7. Anexo: Resultados de la encuesta a desarrolladores).

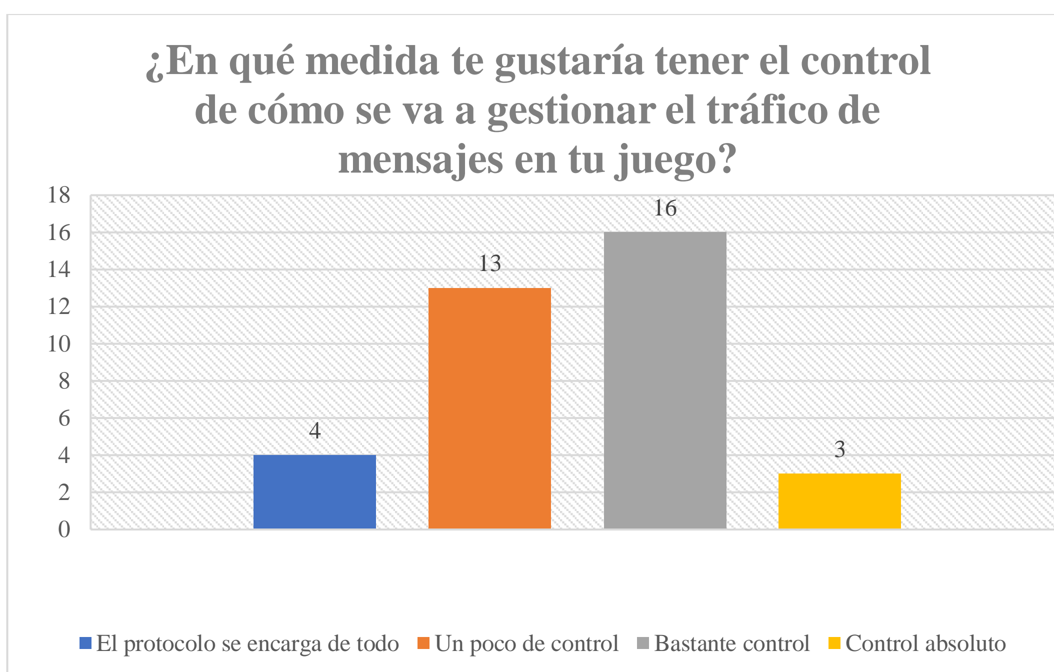

Ilustración 5: Gráfica con los resultados a la primera pregunta de la encuesta a desarrolladores junior.

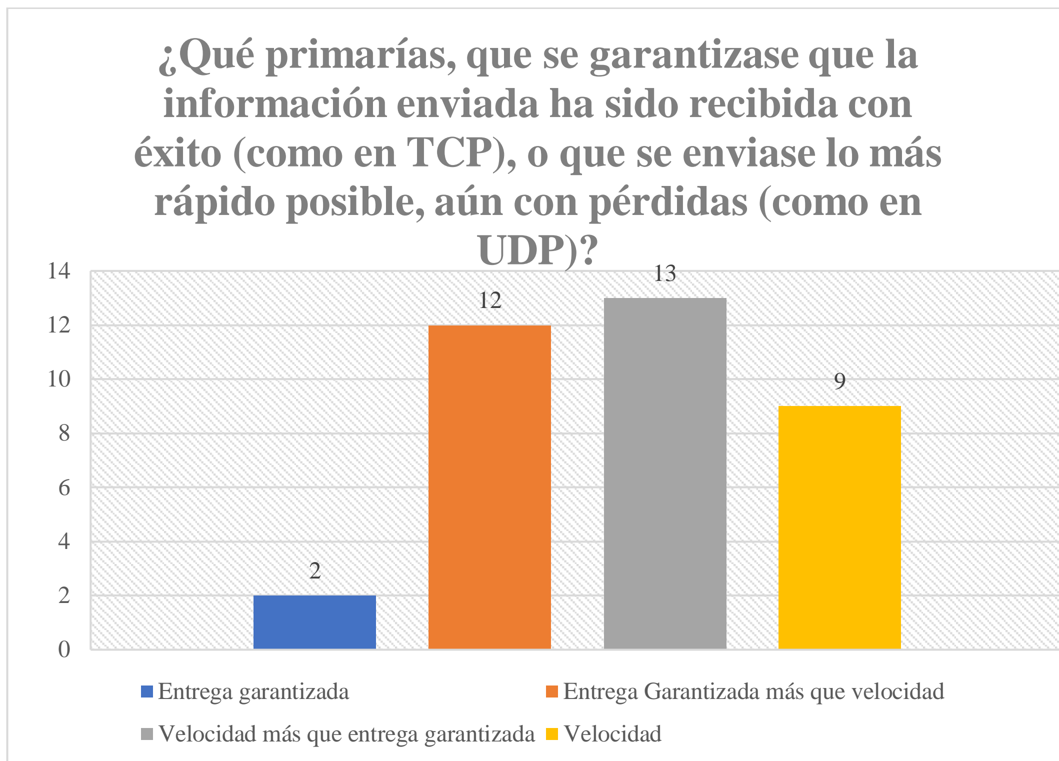

Ilustración 6:Gráfica con los resultados a la cuarta pregunta de la encuesta a desarrolladores junior.

Resumiendo los resultados de la encuesta: los desarrolladores quieren un protocolo que sea cómodo de usar, que sea capaz de tomar decisiones por sí mismo, pero también quieren tener un mínimo de control sobre él (ver Ilustración 5). También ven indispensable que haya algún tipo de mecanismo para poder mandar mensajes que no se puedan perder. También se ve una clara preferencia por una arquitectura cliente-servidor. En cuanto a TPC vs UDP, aunque hay una clara inclinación a priorizar la velocidad (61.1%), pocas personas se atreven a dejar de lado la entrega garantizada, y casi el 70% de los encuestados apuesta por una opción intermedia (ver Ilustración 6) . A raíz de estos resultados, se puede decir que la investigación y desarrollo de los protocolos RUDP están muy ligados al área de los videojuegos.

3.2. Características deseadas

El objetivo principal del desarrollo de esta aplicación será crear un protocolo entre la capa de transporte y aplicación, que de fiabilidad a UDP, sea altamente configurable y esté orientado a su utilización en el entorno de los videojuegos.

Además, y gracias a la información obtenida sobre las preferencias de los usuarios, se pueden especificar otro tipo de características que se quieren implementar en este protocolo:

- Dos modos de entrega: entrega 100% fiable y entrega con porcentaje de pérdidas tolerables. Se diseñará una ventana deslizante adaptada, inspirada en los mecanismos de TCP, con uso de ACK.
- Mecanismos orientados a conexión, protocolos de inicio y finalización de la comunicación.
- Cálculo de RTT y posibilidad de expulsión por medio de un mecanismo de *ping* adaptado.
- Posibilidad de llevar a cabo una retrasmisión rápida (Como en TCP).
- Uso de un *handler* con el que poder personalizar el código a ejecutar para cada tipo de mensaje recibido.
- Mecanismos asociados a la garantía de una recepción ordenada de los paquetes enviados.
- Cómodo, personalizable y fácil de utilizar.
- Sobre UDP para minimizar latencias.
- Orientado a un esquema Cliente-Servidor.

3.2.1. Dos canales de comunicación

En MUSE-RP existirán dos modos de entrega: uno fiable y otro parcialmente fiable. Sin embargo, a diferencia de otros protocolos estudiados, el modo de fiabilidad no irá ligado al paquete en sí, si no al canal utilizado. En otras palabras, cada conexión contará con dos canales: uno con entrega fiable garantizada, y otro con entrega parcialmente fiable garantizada. Cada canal tiene un *socket* y un puerto asociado.

Como se ha adelantado antes, en los videojuegos hay mensajes más importantes que otros. Debe garantizarse la llegada de algunos de ellos, mientras que otros pueden perderse sin problema. Es por ello que se determinó que era justo tratar cada uno de estos tipos de mensajes por separado. Así, aunque el canal parcialmente fiable este saturado, los mensajes prioritarios podrán llegar sin problemas al canal fiable (teniendo en cuenta que este no esté saturado también).

Aunque los mensajes no prioritarios puedan perderse, es bueno poder asegurar un mínimo de entrega fiable. De aquí que el canal no prioritario sea parcialmente fiable. Esta parcialidad se traducirá en que el canal tenga un porcentaje de pérdidas admitidas. Por ejemplo, si se ha configurado para que tenga un 50% de fiabilidad, de N mensajes, podrá perder N/2. Si se han perdido más y no se cumple el porcentaje, actúa como el canal fiable y espera a que el emisor le reenvíe los paquetes.

3.2.2 Mecanismos orientados a conexión

Para que MUSE-RP esté orientado a conexión, se utilizarán mecanismos de inicio y finalización de conexión. Como va a estar orientado a un esquema Cliente-Servidor, será el cliente el que inicie las conexiones, aunque cualquiera de los dos podrá terminarla. Se guardará la información de estas conexiones hasta que se decida cerrarlas. Aun así, estas conexiones pueden cerrarse por un tercer motivo, y es que el otro punto de la conexión no conteste durante un tiempo determinado y configurable.

3.2.2.1 *HandShake*

Ilustración 7: Flujo de paquetes en un evento de inicio de conexión (*HandShake*) en MUSE-RP.

Para establecer una conexión entre dos puntos con MUSE-RP han de establecerse dos canales, cada uno con su propio *socket*: un canal fiable y un canal parcialmente fiable. El proceso es el siguiente (resumido visualmente en la Ilustración 7):

- El cliente manda un paquete *Init* al servidor para iniciar la conexión, con información sobre desde qué número van a comenzar los números de secuencia.
- El servidor responde con otro paquete *Init* con información sobre el canal que va a ser utilizado y sobre cuál es el puerto de su canal parcialmente fiable. Con esto, el servidor da por conectado el canal fiable, aunque hasta que no conecte el segundo canal, el cliente no se registrará como conectado.
- El cliente, al recibir este *Init* configura los dos canales, y manda otro *Init,* esta vez por el canal parcialmente fiable.
- El servidor, al recibir este segundo *Init* por su canal parcialmente fiable, registra la conexión como exitosa, y abre el canal parcialmente fiable. Responde con un cuarto *Init* con información sobre el canal.
- Al recibir este último *Init*, el cliente ya pasa a estar conectado completamente al servidor.

Se diseñó este *Handshake* a base de fallo y error. Al principio se diseñó para que fuese un acuerdo en 3 fases (como en TCP). Sin embargo, el tener que abrir un canal por cada puerto causó problemas, sobre todo por el lado del cliente, ya que en las pruebas el puerto que se registraba como puerto de salida era diferente al especificado, muy posiblemente por que los *router* NAT que se estaban utilizando funcionaban como intermediarios. Por tanto, para una conexión exitosa, se decidió hacer un doble inicio de conexión.

3.2.2.2 Fin de conexión

Los mensajes *End* son más sencillos que los mensajes *Init*. Estos, una vez recibidos por el receptor, cierran la conexión de inmediato. Se mandan siempre que el programador lo requiera, o en caso de *timeout*, por si hay alguna posibilidad de notificarle a la otra parte que se va a desconectar.

Los mensajes *End* no son estrictamente necesarios, ya que mediante el mecanismo expulsión por inactividad, en cuanto se detecte que un host no responde, se elimina la conexión. El que llegue un mensaje de *End* tan solo hace más corto el camino y se ahorra la espera.

3.2.2.3 Ping y cálculo de RTT

Al principio iba a usarse la clase *Ping* (incorporada en C#) para el cálculo de ping y los mecanismos de expulsión en MUSE-RP. Sin embargo se encontró un problema: algunos host no respondían. Esto ocurría también ejecutando el comando *ping* en la terminal para esos mismos host.

Como en MUSE-RP ya se establece un canal de conexión, se decidió crear un mecanismo interno que replicase la funcionalidad de ping. Así es como se creó la clase *PingController,* encargada de manejar los mensajes *ping*.

En cuanto a su funcionamiento, tanto el cliente como el servidor enviarán paquetes ping al otro host. El receptor de este paquete, deberá devolverlo a su dueño. En cuanto le llegue al emisor del paquete la respuesta (marcada por un identificador) calculará el tiempo que ha tardado en responder (el RTT).

Con este RTT, se configurarán también los temporizadores de los canales abiertos con el host. Este tiempo será de 2xRTT, para tener tiempo de margen. En TCP, este tiempo se duplica cuando hay una retrasmisión, pero en el caso de MUSE-RP no se vio necesario.

El intervalo en el que se mandan los pings es configurable, y también se usan para el mecanismo de expulsión. Si un host no responde a X pings (calculados mediante la formula *Tiempo de expulsión/Intervalo de ping*), se cierra la conexión. En muchas ocasiones, los servidores se caen o los clientes se desconectan sin avisar (ya sea por el estado de la red o por otros motivos). Es por ello que tanto el cliente como el servidor pueden cerrar la conexión si la otra parte no responde a los *pings*. Antes de ello siempre mandarán un mensaje de fin de conexión por si acaso llega a su destino.

3.2.3. *Handlers* específicos por tipo de mensaje

Para ayudar al programador a diferenciar sus tipos de mensaje, MUSE-RP cuenta con un mecanismo para añadir *handlers* que manejarán los diferentes paquetes según su identificador de tipo. Este mecanismo ya se vio en el punto 2, ya que muchos protocolos actuales hacen uso de ello, y por ello se decidió que era acertado añadirlo a MUSE-RP.

3.2.4. Comunicación entre capas

Teniendo en cuenta la descripción de la red siguiendo un modelo de capas, el protocolo MUSE-RP actuará como capa intermedia entre la de transporte (UDP) y la de aplicación (el videojuego a desarrollar). La información del mensaje a enviar se moverá entre capas mediante 4 mecanismos diferenciados (representados visualmente en la Ilustración 8):

- *EncapsuleData*: Encapsula los datos en un paquete MUSE-RP, con la cabecera correspondiente.
- *SerializeData*: Pasa a UDP los bytes del paquete para posteriormente enviarlos.
- *DeserializeData:* Crea un paquete MUSE-RP a partir de los bytes que llegan desde la capa de transporte.
- *MessageHandler:* Se pasa el paquete a su *handler* específico para ser procesado. El usuario podrá programar el *handler* para cada tipo de paquete MUSE-RP recibido.

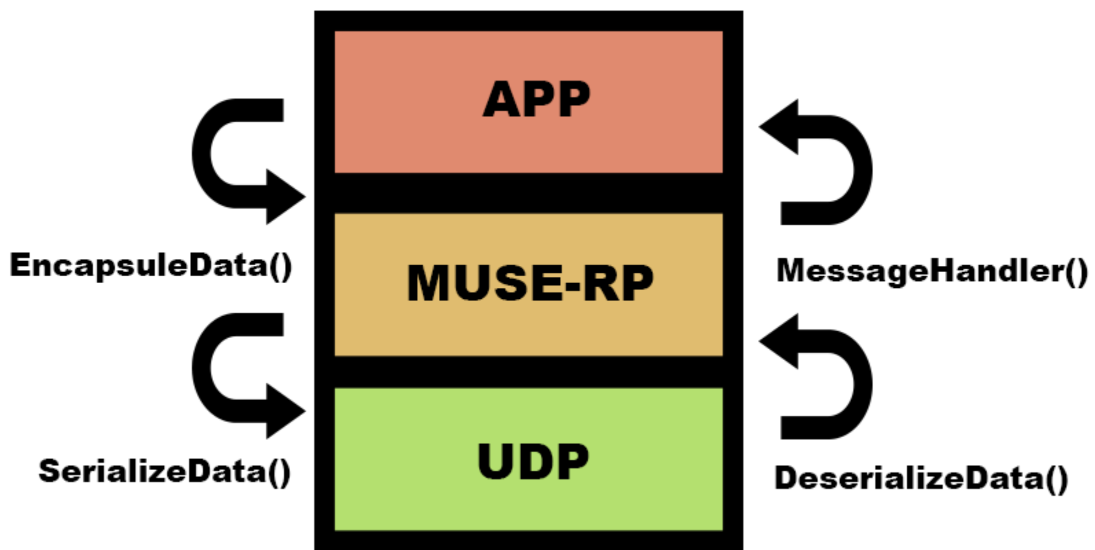

Ilustración 8: Comunicación entre las diferentes capas de red en MUSE-RP

3.3. Metodología

Antes de proceder con el diseño del protocolo, es necesario saber que tecnologías se tienen a disposición. Aunque es igual de importante saber elegir una metodología de desarrollo adecuada que no dificulte la construcción adecuada el protocolo.

3.3.1. Tecnologías usadas

A continuación se ofrecerá una enumeración de las tecnologías utilizadas en el desarrollo de MUSE-RP, así como una breve justificación de su uso.

3.3.1.1 Tecnologías usadas en la implementación del protocolo

Aunque la implementación de MUSE-RP se lleva a cabo en "Implementación de protocolos RUDP en el desarrollo de videojuegos multijugador"[21], se ha debido tener en cuenta la elección del lenguaje a utilizar para su implementación.

Se escogió C# por diversos motivos:

- Es el lenguaje que usa Unity, un motor de videojuegos bastante popular entre los desarrolladores de videojuegos, y que cuenta con una versión gratuita. Para aumentar la compatibilidad del protocolo con este motor, lo ideal era utilizar el mismo lenguaje. También es el que será utilizado en [21].
- Es el lenguaje utilizado por muchos de los protocolos investigados (por ejemplo, Ruffles o GServer).
- No es un lenguaje a muy bajo nivel, lo cual agilizará el desarrollo de la aplicación.
- Cuenta con varias librerías y APIs muy útiles a la hora de utilizar *sockets* y otro tipo de herramientas relacionadas con las redes.

También se decidió utilizar como repositorio *GitHub*, donde finalmente el protocolo sería subido y compartido con la comunidad cómo código abierto.

Por último, se ha usado *WireShark* tanto para la investigación previa al diseño de MUSE-RP (ver 2.1. Casos de estudio), como para ayudar en el desarrollo del protocolo. Esto es debido a que el envío y recepción de paquetes muchas veces es transparente al

usuario, dificultando las tareas de testeo, cosa que soluciona *WireShark* con su herramienta de captura de paquetes.

3.3.1.2 Tecnología usada en el diseño del protocolo

Para el diseño del protocolo se han utilizado herramientas tales como *StarUML* (para la creación de diagramas de clases *UML*) y *Photoshop* (para la creación de diagramas de flujo de paquetes).

3.3.2. Metodología de desarrollo

En cuanto a la metodología utilizada para el desarrollo de MUSE-RP, primero ha de tenerse en cuenta que a medida que se fue diseñando, se fue implementando [21]. Este detalle es muy importante, porque marcaría el ritmo del desarrollo.

Para el desarrollo del protocolo se han tenido que hacer muchas pruebas, y el resultado final fue fruto de muchas iteraciones y del aprendizaje a base de fallo y error. Esto es debido a que las redes son un entorno impredecible y dado a provocar fallos de diversos tipos. Es por ello que utilizar metodologías tales como el desarrollo en cascada[20] hubiera sido contraproducente. En este tipo de desarrollos se debe de tener muy claro el diseño para poder proceder con la implementación final. De haber seguido dicha filosofía, el producto resultante hubiera tenido deficiencias y varios apartados muy mejorables.

Por otra parte, utilizar metodologías ágiles[20] que pudiesen adaptarse a los cambios, derivados de problemas inesperados que pudieran surgir en el desarrollo parecía lo más óptimo. Además, también se permitiría añadir nuevos requisitos al diseño sin mayor problema.

No se siguió un tipo de metodología ágil específica, sino más bien una metodología híbrida con características de Scrum, adaptada a la ausencia de un equipo de desarrollo y de un cliente. Esta metodología siguió los siguientes pasos:

- Creación de metas o requisitos alcanzables para cada iteración, fruto del análisis del diseño inicial y de las prioridades de cada funcionalidad.
- Implementación siguiendo el diseño acordado.

- *Testing* de cada funcionalidad nueva implementada en un entorno real (en este caso, la red) , creando una lista de mejoras y posibles problemas que dificulten el desarrollo de la aplicación.
- Investigación sobre posibles soluciones y mejoras derivadas del paso anterior.
- Rediseño y adaptación del diseño inicial para poder solucionar los problemas surgidos en el *testing*. También podrá acomodarse a posibles mejoras detectadas en dicha fase.
- Repetición desde el primer paso.

4. Diseño de MUSE-RP

Ya se tiene una investigación lo suficientemente sólida como para comenzar a diseñar un nuevo protocolo RUDP. En este apartado se irán enumerando las características del protocolo, así como la manera en que se piensa implementarlas.

4.1. Algoritmos de entrega fiable

Con el mecanismo de ventana deslizante de TCP en mente, se ajustó el diseño de sus algoritmos para poder cumplir con todos los requisitos que se habían especificado para MUSE-RP.

En cuanto al algoritmo del emisor (es decir, las comprobaciones que tendrá que hacer un Host al llegarle un mensaje) será el mismo para los dos casos, ya que tendrá que comprobar que el ACK recibido es correcto, los ACK duplicado, el fin de temporización y aumentar el número de secuencia por cada mensaje enviado. Ninguna de estas comprobaciones se modifica al no garantizar fiabilidad. De hecho, si se implementara un algoritmo sin entrega fiable, pero garantía de entrega ordenada, los mecanismos del emisor serían los mismos.

A continuación se mostrarán todos estos algoritmos diseñados en pseudocódigo. Pero antes, han de establecerse una serie de variables y su correspondiente nomenclatura (además de unas cuantas aclaraciones):

V: Tamaño de ventana

P: Porcentaje de pérdidas admitidas

S^N: Número de secuencia del paquete a procesar

S^A: Último número de secuencia reconocido

A^N: Número de ACK a procesar

A^A: Último ACK/paquete reconocido

C: Número de conteo actual

F^A: Fallos acumulados actuales

F^N: Cálculo de fallos acumulados provisional

F^M: Máximo número de fallos admitido

D: Distancia desde el paquete procesado al que se está procesando

$F^M = (V x P)/100$ (Redondeado)

$D = S^N - S^A + 1$

$F^N = F^A + D$

$C = C + D + 1$ (Cuando se actualiza)

4.1.1. Algoritmos Emisor

El emisor debe encargarse de 3 cosas: enviar paquetes(Algoritmo 1), comprobar los ACK recibidos(Algoritmo 2) y reenviar un mensaje si se dispara el temporizador.

Estos algoritmos serán iguales para ambos modos de fiabilidad, ya que tanto para la entrega fiable como para la parcialmente fiable, este debe tener una ventana de envío y debe comprobar los ACK que recibe (y reenviar paquetes en caso de un evento de fin de temporización).

```
void Enviar(paquete)
{
        if(paquete.sequence - sequence <= V)
        {
                if(TemporizacionStopped)
                {
                        RestartTemporizacion()
                }
                Enviar(paquete)
                BufferEnviados.Enqueue(paquete)
        }
        else
        {
                BufferEspere.Enqueue(paquete)
        }
}
```

Algoritmo 1: Algoritmo de envío de paquetes del emisor

```
void RecibirAck(newACK){
        if(newACK > currentACK && newACK - currentACK <= V)
        {
                BufferEnviados.RemoveInterval(0,newACK - currentACK)
                currentACK = newACK
                conteoDuplicados=0
                PararTemporizacion()
                if(BufferEspera.Empty()
                {
                        NuevoEnvio = BufferEspera.Dequeue()
                        Enviar(NuevoEnvio)
                }
        }
        else if(newACK == currentACK)
        {
                conteoDuplicados++
                if(conteoDuplicados == 3)
                { //retrasmisión rápida
                        conteoDuplicados = 0
                        RestartTemporizacion()
                        Enviar(BufferEnviados.Dequeue())
                }
        }
}
```

Algoritmo 2: Algoritmo de procesado de paquetes ACK del emisor

4.1.2. Algoritmo receptor entrega fiable

Con el funcionamiento del algoritmo de ventana deslizante en mente, debe tenerse en cuenta lo que se quiere lograr con MUSE-RP. De los estudiados, el algoritmo TCP es el que más se adapta a las necesidades del protocolo, aunque se modificará para acomodarse al mismo. Por ejemplo, ya se adelantó antes que los números de secuencia no corresponderían con los bytes enviados, si no que irán incrementándose de uno en uno, como GBN o SR. Aunque también almacenará los paquetes desordenados, para su posterior procesado en caso de recibir los paquetes que faltan para que les llegue su turno. Este buffer deberá ser capaz, internamente, de ordenar los paquetes por su número de secuencia. Así, cuando se vaya a consultar los paquetes almacenados, se consultarán por orden. Usará, como GBN y TCP, ACKs acumulativos, por tanto no tendrá un reconocimiento selectivo. Por último, al igual que TCP, contará con un mecanismo de retrasmisión rápida al llegar 3 ACKs duplicados.

```
void Procesado(paquete)
{
        newSequence = paquete.sequence
        if(newSequence > currentSequence + 1 )
        {
                if(newSequence - currentSequence <= V)
                {
                        BufferRecibidos.Enqueue(paquete)
                }
                else
                {
                        //Se descarta el paquete
                }
        }
        else if (newSequence == currentSequence + 1)
        {
                currentSequence = newSequence
                PasarACapaSuperior(paquete)
                if(!BufferRecibidos.Empty())
                {
                        Procesado(BufferRecibidos.Dequeue())
                }
        }
        else
        {
                //Se descarta el paquete
        }
        MandarACK() //Ya haya sido actualizado o no
}
```

Algoritmo 3: Procesado de paquetes recibidos por el receptor en el modo fiable de MUSE-RP

En el Algoritmo 3 se puede observar que es un método recursivo. Si el paquete es el que se esperaba, se comprueba que no queden paquetes esperando en el buffer, procesándolos hasta llegar a uno que no corresponda al siguiente esperado, o a que el buffer esté vacío. Al consistir en un buffer ordenado, esta tarea se vuelve más sencilla.

Para una demostración más visual, observar la Ilustración 9 con un caso de prueba de cómo deberá comportarse este algoritmo.

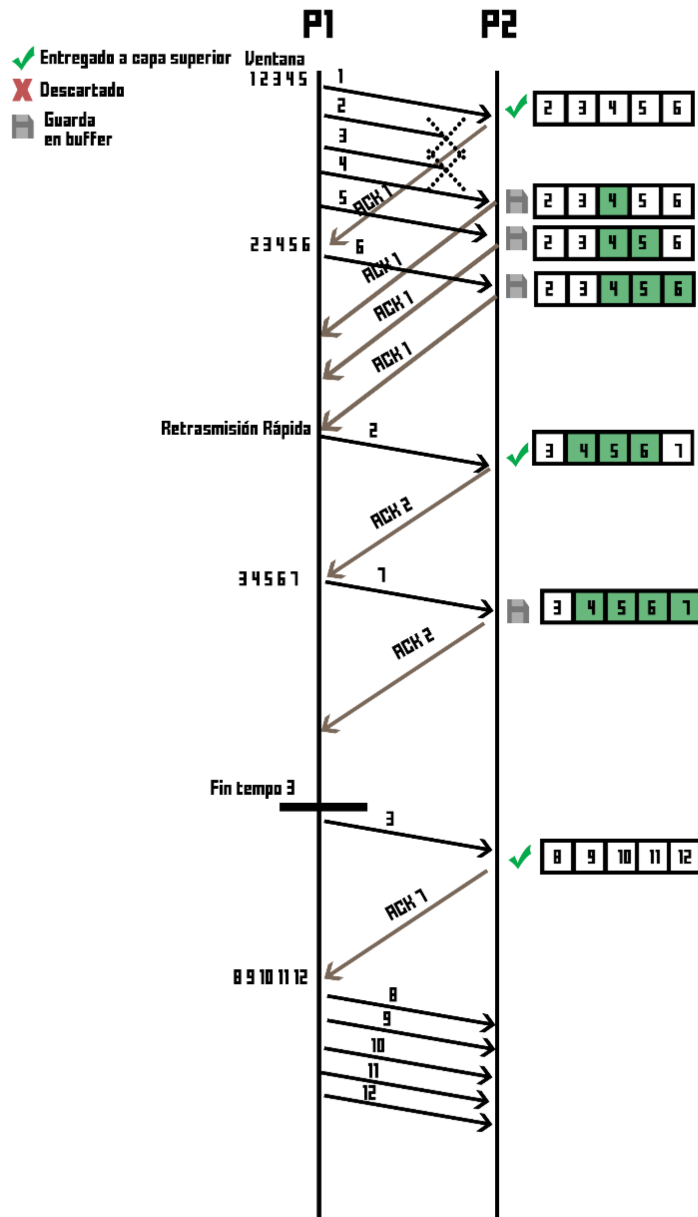

Ilustración 9: Flujo de paquetes entre dos host mediante el algoritmo fiable de MUSE-RP, con un tamaño de ventana de 5 paquetes

4.1.3. Algoritmo receptor entrega parcialmente fiable

Para conseguir un algoritmo de entrega parcialmente fiable, se debe tener como base el algoritmo de entrega fiable, y a partir de ahí, ajustarlo. Para empezar, hay más parámetros a tener en cuenta : el número de fallos permitidos por cada N paquetes, y el número de fallos actuales. Para ahorrarle al desarrollador la elección de ese número N, el protocolo utiliza el tamaño de ventana.

Garantizar un límite en el máximo de pérdidas de paquetes no es tarea sencilla. Utilizar porcentajes y máximo de pérdidas por N paquetes ayuda a simplificarlo. Pero, ¿Cómo trasladar esto al algoritmo? La solución que se propone es la siguiente: el receptor dará por válido cualquier mensaje con un número de secuencia que cumpla las siguientes condiciones:

- Que sea mayor al número de secuencia del último paquete reconocido
- Que la diferencia no sea mayor al tamaño de ventana
- Que la distancia entre el paquete esperado(es decir, S^A+1) y el recibido, sumada con las perdidas ya acumuladas, no supere el máximo de pérdidas.

Véase con un caso de ejemplo: en una conexión se tiene un tamaño de ventana de 10 y un porcentaje de pérdidas tolerables del 20% (es decir, 2 paquetes por cada 10). Han llegado los mensajes 1, 2 y 4. Al llegar el 4, el algoritmo detecta que la distancia con el paquete esperado (3) es 1, por tanto se ha perdido un paquete. Hay 0 fallos acumulados, 0+1<=2, que es el máximo de fallos admitidos por ventana. Eso quiere decir que el paquete puede pasar a capa superior sin ningún problema (mandando su correspondiente ACK reconociendo el paquete 4). Si ahora llegase el paquete 6 también pasaría sin problemas, ya que 1+1<=2. Sin embargo, si ahora llegase el paquete 8, el algoritmo no lo reconocería, dado que las pérdidas superan el valor deseado (2+1>2). En este caso, actuaría como el algoritmo fiable, guardando el paquete y mandando un ACK (en este caso, ACK al 6) sin actualizar el número del último paquete reconocido.

Aunque este método funciona para la mayoría de los casos, hay casos excepcionales en los que podría no cumplir con el máximo de perdidas garantizado. Por ejemplo: en un caso hipotético de un flujo de información donde se mandan un total de 50 mensajes, el tamaño de ventana es 100, y el porcentaje de pérdida es de un 50%, este método podría causar un porcentaje de pérdidas mayor que el indicado. Si solo se recibe el paquete 1 y luego el 50, el algoritmo reconocería el paquete 50 sin problemas ya que 50-2 = 48; 48<=50 (aunque no se podría permitir tener muchas más perdidas). Si la comunicación termina ahí, se habrá tenido unas pérdidas del 96%. Sin embargo, este escenario es poco probable dado que en los videojuegos hay un flujo continuo de datos. También se tiene que tener cuidado al seleccionar el tamaño de ventana.

El algoritmo, además, deberá tener en cuenta el cambio de ventana. Por ejemplo, con un tamaño de ventana de 10, los paquetes del rango (1,10) serán parte de una ventana

diferente a la del rango (11,20), y así sucesivamente. El algoritmo debe de estar preparado para los casos en los que los paquetes a reconocer pasan de formar parte de una ventana a otra, ya que aunque la ventana anterior tenga X pérdidas, esta nueva ventana comenzará con 0 fallos.

En el Algoritmo 4 puede observarse el resultado del diseño del algoritmo en cuestión.

```
void Procesado(paquete)
{
        newSequence = paquete.sequence
        D= newSequence - (currentSequence +1)
        if(D < 0)
        {
                //Se descarta el paquete
        }
        else if(D == 0)
        {
                C++
                if(C >=V)
                {
                        C = 0
                        currentFails = 0
                }
                currentSequence = newSequence
                PasarACapaSuperior(paquete)
                if(!BufferRecibidos.Empty)
                {
                        Procesado(BufferRecibidos.Dequeue())
                }
        }
        else //D>0
        {
                if ( D + C + 1 > V)  //Comprobaciones si se trata de un
                {                    //cambio de ventana
                        H1 = V - C
                        H2 = D - H1
                        newFails = currentFails + H1
                        if(newFails <= maxFails && H2 <= maxFails)//
                        {
                                C = H2 + 1
                                currentFails = H2
                                currentSequence = newSequence
                                PasarACapaSuperior(paquete)
                                If(!BufferRecibidos.Empty)
                                {
                                        Procesado(BufferRecibidos.Dequeue())
                                }
                        else
                        {
                                BufferRecibidos.Add(PaqueteActual)
                        }
                }
                else
                {
                        newFails= currentFails + D
                        if( newFails > maxFails)
                        {
                                BufferRecibidos.Add(PaqueteActual)
                        }
                        else
                        {
                                currentFails = newFails
                                C = C + D +1
                                currentSequence = newSequence
                                PasarACapaSuperior(paquete)
                                if(C==V)
                                {
                                        C = 0
                                        currentFails = 0
                                }
                                if(!BufferRecibidos.Empty)
                                {
                                        Procesado(BufferRecibidos.Dequeue())
                                }
                        }
                }
        }
        MandarACK();
}
```

Algoritmo 4: Procesado de mensajes recibidos por el receptor en el modo parcialmente fiable de MUSE-RP

Para una mayor comprensión de este algoritmo, a continuación se exponen tres posibles casos de flujo de paquetes en una conexión que aplique el algoritmo de entrega parcialmente fiable (Ilustración 10, Ilustración 11 e Ilustración 12).

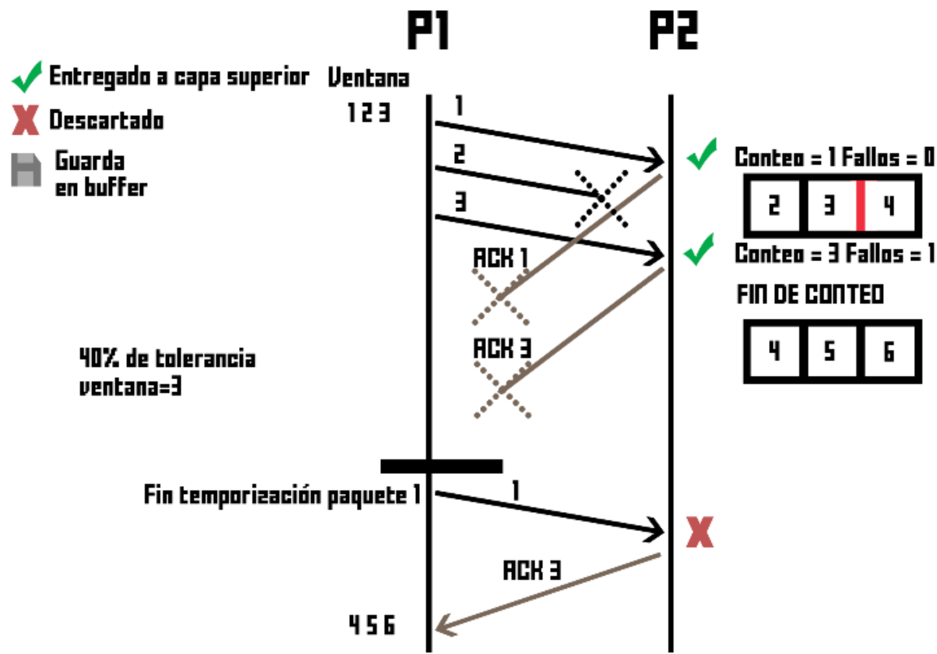

Ilustración 10: Flujo de paquetes entre dos host con el algoritmo parcialmente fiable de MUSE-RP, donde el tamaño de ventana es de 3 y el porcentaje de fiabilidad 60%

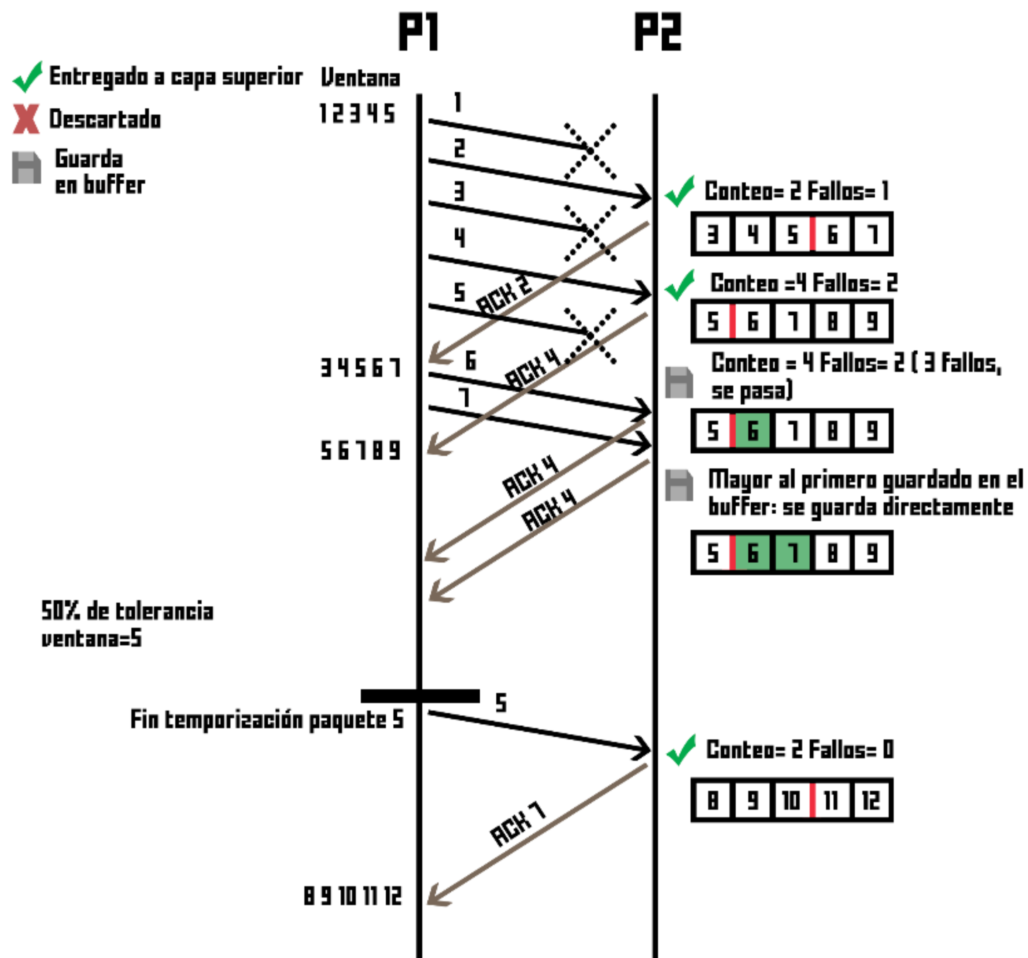

Ilustración 11: Flujo de paquetes entre dos host con el algoritmo parcialmente fiable de MUSE-RP, donde el tamaño de ventana es de 5 y el porcentaje de fiabilidad 50%

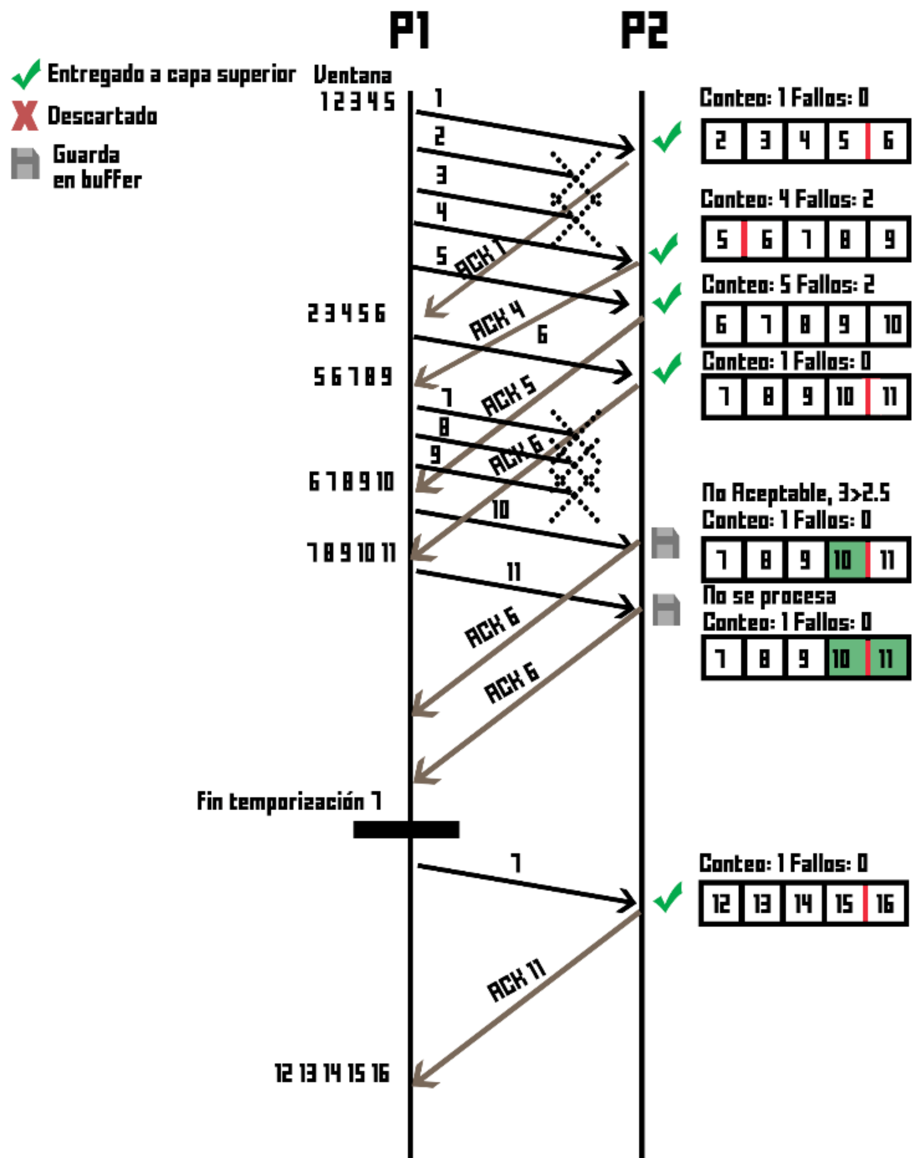

Ilustración 12: Flujo de paquetes entre dos host con el algoritmo parcialmente fiable de MUSE-RP, donde el tamaño de ventana es de 5 y el porcentaje de fiabilidad 50%

4.2. Estructura de un paquete MUSE-RP

Los protocolos en redes suelen tener una estructura definida para el tipo de datos que se envían, y en el caso de MUSE-RP, esta estructura estará encapsulada dentro de la carga útil de un paquete UDP.

4.2.1. Estructura

Un paquete MUSE-RP cuenta con una cabecera de 11 bytes:

- Un byte está reservado para los *flags* ACK, PING, END e INIT.
- Dos bytes están reservados para un identificador de tipo de paquete (un *ushort* en C#).
- Cuatro bytes están reservados para el número de secuencia.
- Los últimos cuatro bytes están reservados para el número de ACK.

En la Ilustración 13 puede verse la distribución de todos estos campos dentro del paquete.

Ilustración 13: Esquema de la estructura de un paquete MUSE-RP

4.2.2. *Flags*

MUSE-RP cuenta con 4 *flags* fundamentales. Cada una ocupa un bit, por tanto solo pueden tomar el valor 0 o 1, es decir, “verdadero” o “falso”. Aunque con 4 bits hubiera bastado para este objetivo, se utiliza un byte entero para poder operar con la cabecera lo más cómodo posible.

Las *flags* son las siguientes:

- **ACK**: indica que el mensaje es un mensaje de reconocimiento, es decir, que no hay carga útil en el mensaje, si no que el receptor de este paquete debe analizar el campo de ACK para hacer los cálculos correspondientes de reconocimiento de mensajes.
- **PING**: indica que el paquete es un mensaje ping, es decir, que debe delegarse al controlador del mecanismo de ping. En este caso el paquete sí que llevará carga útil: un booleano que indicará si el paquete ping es del emisor o del receptor de dicho paquete (es decir, si es un ping que envió y ha sido devuelto, o si es un paquete ping enviado por otro host).
- **INIT**: indica que el paquete es un mensaje INIT. Su funcionamiento ya se vio en el apartado 3.2.2.1 *HandShake*, y este varía según el que canal en el que ha sido recibido y el tipo de host que es.
- **END**: indica que el paquete es un mensaje END, y como se indica en el apartado 3.2.2.2 Fin de conexión, esto cerrará las conexiones del emisor correspondientes al emisor del mensaje.

4.2.3. Campos

Un paquete MUSE-RP cuenta con 10 bytes para 3 campos: número de secuencia, número de ACK y tipo de mensaje. Desde el punto de vista del código, los dos primeros serán *uint* (es decir, enteros positivos de 4 bytes), y el último un *ushort* (un entero positivo de 2 bytes). Esto quiere decir que los números de secuencia están comprendidos en un rango del 0 al 2^{32} (4.294.967.296).

- **Número de secuencia**: identifica al mensaje, lo cual ayudará al receptor del mismo a ejecutar su algoritmo de ventana deslizante y concluir si debe almacenar el mensaje, pasarlo a capa superior o descartarlo.
- **Número de ACK**: campo que se examina en caso de tratarse de un paquete ACK para ejecutar el algoritmo de reconocimiento.
- **Identificador de tipo**: que, al pasar el paquete a la capa de aplicación, determinará que *handler* ejecutar para su procesamiento. Al tratarse de un *ushort*, podrán definirse hasta 2^{16} (65.536) tipos de mensajes.

4.3. Diseño del código

Ya se han establecido las características fundamentales de MUSE-RP, ahora es el momento de diseñar el código y sus clases. Esto se hará teniendo en cuenta, en todo momento, que se va a diseñar para el lenguaje C# y el *framework* .NET.

4.3.1 Descripción de clases

Se dará en esta sección una breve descripción de la estructura y responsabilidades de cada una de las clases principales diseñadas para la implementación de MUSE-RP.

Host

La clase abstracta *Host* representa un punto terminal en la red. De ella heredarán tanto la clase *Client* como la clase *Server.* Se encargará de guardar las conexiones abiertas, de tener una referencia a los *SocketHandler* que gestionarán los canales, de gestionar la configuración del punto terminal y de procesar los mensajes que lleguen.

Entre sus métodos se pueden encontrar métodos de envío y recepción (aunque no se encargue directamente de ello), de intento de conexión, de inicialización y de gestión de *handlers*. Además de todo esto, cuenta con la posibilidad de crear un hilo donde gestionar los mensajes que va recibiendo en un buffer de procesado.

Dentro de la estructura de las capas de la red, esta clase se encontraría en la capa de aplicación, ya que es la encargada de gestionar los mensajes que ya han pasado por todas las comprobaciones de fiabilidad.

Client

Especificación de la clase *Host*. Entre sus métodos específicos pueden encontrarse métodos de envío al servidor o métodos orientados a intentar conectarse con él. También cuenta con métodos auxiliares de generación aleatoria de puertos para conectarse y números de secuencia para la fase de inicio de conexión.

Contará con un *SocketHandler* específico para el inicio de conexión y un *PingController*, y especificará los eventos de Ping, End, Init e inicialización del *host*.

Server

Especificación de la clase *Host*. A diferencia del cliente, el servidor no tiene métodos específicos de intento de conexión, ya que al inicializarse solo necesitará escuchar en los sockets indicados. Cuenta con varios *PingControlle*r (uno por cada cliente) y con especificación en los *handlers* de eventos de Ping, End e Init.

HostOptions

Esta estructura guarda todas las especificaciones de un *Host*: conexiones permitidas, tiempo de *TimeOut*, intervalo de los mensaje ping, tick de procesado de mensajes, número de puertos (tanto el fiable como el no fiable), tamaño de ventana, tiempo de temporización inicial, porcentaje de fiabilidad del canal no fiable y un *MessageHandler*. También cuenta con un único método, *HandleMessage(),* que invoca al *MessageHandler* para que maneje un mensaje recibido.

Connection

Esta clase almacena la información de un *Host* de una conexión y canal específicos: su ID, su dirección IP, su puerto, si es fiable o no y, en caso de necesitarlo, el EndPoint al que hace referencia.

ConnectionInfo

A diferencia de *Connection, ConnectionInfo* guarda información sobre la IP y los dos puertos utilizados (tanto el fiable como el no fiable) de una conexión con un punto terminal específico.

PingController

Esta es la clase encargada de mandar pings, además de que será invocada cada vez que se reciba uno. Tiene información sobre el intervalo de ping especificado, el host operado en la máquina, y el canal al que hace referencia.

Llevará un conteo de los pings no respondidos, además del ID del ping que espera recibir. Si le llega un ping antiguo y no es el que espera, este no se tendrá en cuenta. Esto se puede traducir en que si el ping es mayor al intervalo de envío de pings, la conexión acabará cerrándose por *timeout*.

MessageHandler

Un objeto *MessageHandler* contiene un diccionario de delegados especiales (*MessageDelegate*), cuyas claves son *ushorts* que representan cada tipo de mensaje. Estos delegados especiales esperan tanto un *MessageObject* como una *Connection* con la información del origen del mensaje. En un *MessageHandler* pueden añadirse, eliminarse e invocarse dichos delegados.

MessageObject

Estructura que almacena toda la información de un mensaje MUSE-RP (tanto la cabecera como los datos). Recordando el apartado 4.2. Estructura de un paquete MUSE-RP, esta estructura cuenta con: un byte de *flags*, un *ushort* con el tipo de mensaje, un uint para el número de secuencia y otro para número de ACK, y un array de bytes con la carga útil del mensaje.

También cuenta con métodos para pasar toda esta información a bytes, así como para convertir un array de bytes a un objeto *MessageObject*. Dichas conversiones no se hacen implícitamente para poder controlar mejor el tamaño en bytes del mensaje a enviar.

ChannelInfo

Estructura con la información necesaria sobre un canal: puertos fiable y no fiable, tamaño de ventana, porcentaje de fiabilidad, puertos del *Host* propio e ID. También cuenta con métodos específicos para guardar la información en forma de bytes, ya que esta clase se utilizará para que el servidor y el cliente pueden comunicarse información sobre los canales que se van a utilizar.

SocketHandler

Clase auxiliar ligada a cada puerto utilizado por la máquina en las múltiples conexiones que pueda tener. Cada *Host* tendrá dos, uno por cada canal.

Los *SocketHandler* se encargan de recibir los mensajes que llegan al puerto al que están ligados, redirigiéndolos al canal correspondiente según su procedencia. Contarán con un hilo de recepción y otro de procesado, además de una referencia al *socket* en el que escucha.

También son los encargados de mandar los mensajes pasados por la capa de aplicación por el canal correspondiente. Además, al tener varios canales a su cargo, serán los encargados de gestionarlos.

ChannelHandler

Clase abstracta que hace referencia a una generalización de una clase canal. Actualmente tiene dos implementaciones: una para los canales fiables y otra para los canales parcialmente fiables.

Cuenta con información esencial sobre el canal : su conexión, el tamaño de ventana, el tiempo de reenvío, información sobre su estado (como el número de secuencia actual, el número de ACK actual, o el número de último número de secuencia reconocido). Además, contará con 4 buffers con información sobre los mensajes a enviar, enviados, recibidos y a la espera de ser enviados. Diferenciar los mensajes a enviar de los que están esperando a ser enviados, ya que los primeros son los que van añadiéndose cada vez que un *Host* ejecuta el comando *Send*, mientras que los segundos son los que están a la espera de que la ventana de envío tenga hueco.

Es en esta clase donde ya se implementa un envío funcional, además de la gestión de los finales de temporización y ACKs recibidos. Podría decirse que se encuentra en la capa intermedia entre UDP y la aplicación.

También cuenta con un método abstracto *Receive,* el cual será especificado por cada implementación de esta clase. Será este el que decida qué mensajes pasan a capa superior, cuales almacenar en el buffer y cuales descartar.

PartialReliableHandler

Especificación de C*hannelhandler* orientada a un canal fiable. Su método de recepción se inspira en el Algoritmo 3: Procesado de paquetes recibidos por el receptor en el modo fiable de MUSE-RP.

ReliableChannelHandler

Especificación de *ChannelHanlder* orientada a un canal parcialmente fiable. Añade variables tales como el número de conteo, el conteo de fallos actuales y el máximo de fallos permitidos. El método de recepción se inspira en el Algoritmo 4: Procesado de mensajes recibidos por el receptor en el modo parcialmente fiable de MUSE-RP.

MuseBuffer

Encapsulación de una cola de elementos genéricos que actúa como buffer que se utilizará para almacenaje de mensajes en diversas encapsulaciones. Cuenta con un objeto *Mutex* que encapsula cada operación que implique a la cola, reduciendo así posibles fallos de concurrencia consecuencia del uso de varios hilos.

Cuenta con métodos para sacar elementos del buffer, conseguir el primer elemento, añadir elementos, añadir elementos únicos (es decir, siempre y cuando no estén ya añadidos), limpiar el buffer, y eliminar una cierta cantidad de elementos.

SortedMuseBuffer

Encapusulación de un *SortedSet* de elementos genéricos que actúa como buffer que se utilizará para almacenaje de mensajes en diversas encapsulaciones. En esencia, contiene la misma funcionalidad que un *MuseBuffer*. La única diferencia está en el uso de un *SortedSet* en vez de una cola convencional.

Se necesita este tipo de buffer en MUSE-RP ya que algunos buffers necesitan estar ordenados para su correcto funcionamiento. Es el caso, por ejemplo, de los buffers de recepción de los algoritmos fiables. Estos deberán estar ordenados para poder recorrerlos de una manera más optima en caso de que llegue un paquete que rellene algún hueco. Se hizo un análisis de que tipo de estructuras de datos ordenada se debería utilizar, siendo *SortedSet* la que más se adaptaba a las necesidades del diseño sin necesidad de perjudicar mucho al rendimiento.

4.3.2 Diagramas UML

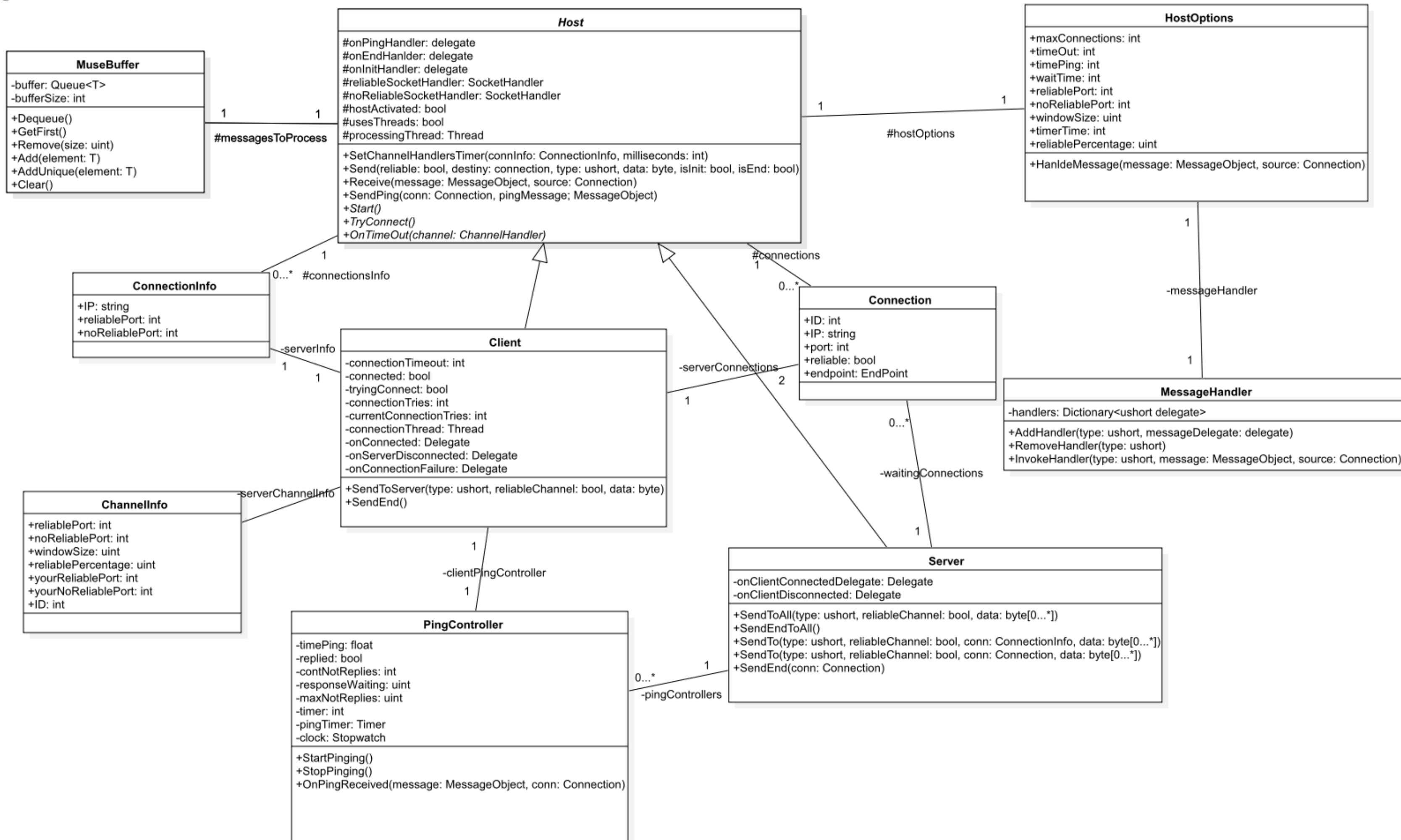

Ilustración 14: Diagrama UML de las clases relacionadas con los puntos terminales del protocolo MUSE-RP (Capa de aplicación)

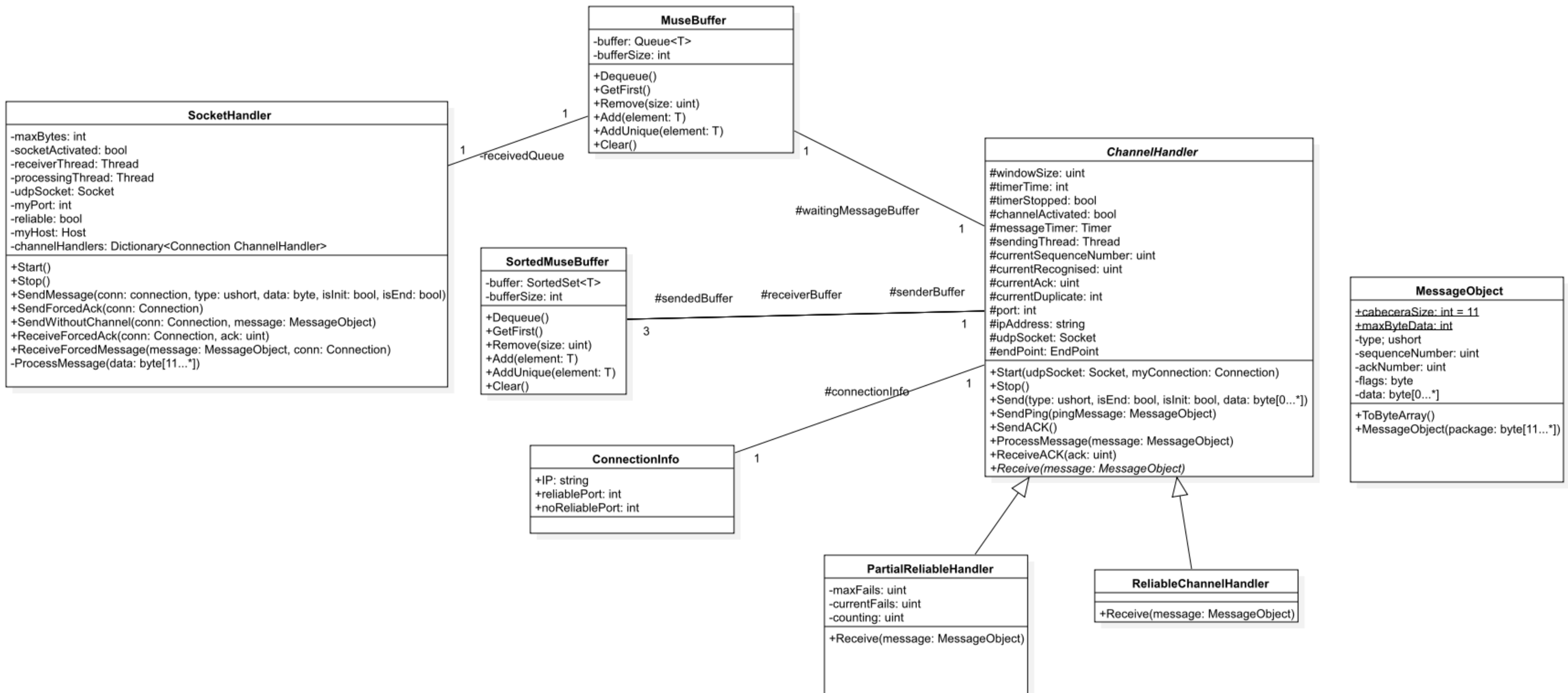

Ilustración 15:Diagrama UML de las clases relacionadas con los canales de comunicación del protocolo MUSE-RP (Capa intermedia)

Los diagramas UML, Ilustración 14 e Ilustración 15, son el resultado final de muchas iteraciones en el desarrollo de MUSE-RP. Difieren bastante del diseño de clases original, ya que muchas clases fueron adaptándose a los requisitos reales de la aplicación. Se ha querido dividir en dos para una mejor comprensión. La Ilustración 14 hace referencia a todas las clases relacionadas con la capa de aplicación y los puntos terminales de la red, mientras que la Ilustración 15 hace referencia a clases de más bajo nivel, relacionadas con los canales de comunicación, los *sockets* y los algoritmos de fiabilidad.

Destacar que estos diagramas están resumidos y que no incorporan todos los métodos de cada clase, ya que dificultaría su lectura. Se han priorizado aquellos que son públicos e indispensables para la utilización del protocolo. Para una visión más concreta del protocolo, visitar: https://github.com/Celtia-Martin/MUSE-RP-RUDP-protocol

Esta implementación fue fruto del trabajo "Implementación de protocolos RUDP en el desarrollo de videojuegos multijugador"[21].

4.4. Pruebas y resultados

Antes de implementar el diseño completo de MUSE-RP, se hicieron varias pruebas de concepto sobre su característica más importante: la fiabilidad. Se implementó el algoritmo parcialmente fiable en Python y se hicieron varias pruebas para comprobar que funcionaba tal y como estaba previsto (el código está disponible en el 8. Anexo: Código en Python de prueba del algoritmo parcialmente fiable.

El programa de prueba espera, primero, una entrada de dos números: el tamaño de ventana y el porcentaje de fallos tolerables. Después, espera a la llegada de paquetes, o más bien, de números de secuencia, empezando por el paquete 1. Con la llegada de estos números de secuencia, el programa devolverá una salida con información sobre el estado del receptor: el ACK que se devuelve, el buffer de paquetes recibidos, cuantos fallos lleva en el conteo actual, el conteo actual y el porcentaje de pérdidas totales.

```
C:\WINDOWS\py.exe
10 50
Ventana: 10 Porcentaje: 50.0 Máximo de fallos tolerables: 5
1

Se devuelve: 1
El conteo esta en: 1
Los fallos a:0
Y el buffer: []

Porcentaje de pérdidas: 0.0
8

Se guarda 8 en buffer

Se devuelve: 1
El conteo esta en: 1
Los fallos a:0
Y el buffer: [8]

Porcentaje de pérdidas: 0.0
6

Se devuelve: 8
El conteo esta en: 8
Los fallos a:5
Y el buffer: []

Porcentaje de pérdidas: 62.5
10

Se guarda 10 en buffer

Se devuelve: 8
El conteo esta en: 8
Los fallos a:5
Y el buffer: [10]

Porcentaje de pérdidas: 62.5
11

Se guarda 11 en buffer

Se devuelve: 8
El conteo esta en: 8
Los fallos a:5
Y el buffer: [10, 11]

Porcentaje de pérdidas: 62.5
12

Se guarda 12 en buffer

Se devuelve: 8
El conteo esta en: 8
Los fallos a:5
Y el buffer: [10, 11, 12]

Porcentaje de pérdidas: 62.5
9

Se devuelve: 12
El conteo esta en: 2
Los fallos a:0
Y el buffer: []

Porcentaje de pérdidas: 41.66666666666667
13

Se devuelve: 13
El conteo esta en: 3
Los fallos a:0
Y el buffer: []

Porcentaje de pérdidas: 38.46153846153847
14

Se devuelve: 14
El conteo esta en: 4
Los fallos a:0
Y el buffer: []

Porcentaje de pérdidas: 35.714285714285715
15

Se devuelve: 15
El conteo esta en: 5
Los fallos a:0
Y el buffer: []

Porcentaje de pérdidas: 33.33333333333333
20

Se devuelve: 20
El conteo esta en: 0
Los fallos a:0
Y el buffer: []

Porcentaje de pérdidas: 45.0
25
```

Ilustración 16: Salida por consola a un caso de prueba de comprobación de números de secuencia para el algoritmo parcialmente fiable.

En la Ilustración 16 puede verse un caso de prueba de dicho algoritmo: el tamaño de ventana será de 10 paquetes, con un máximo de un 50% de pérdidas tolerables. La entrada de paquetes será : 1, 8, 6, 10, 11, 12, 9, 13, 14, 15, 20, 25. Como se previó en apartados anteriores, hay ocasiones en los que el porcentaje de pérdida puede llegar a ser mayor del esperado (por ejemplo, con la llegada del paquete 8). Esto es debido a que, aunque se ha llegado al máximo de fallos por conteo (5), el conteo aún no se ha completado (el último paquete reconocido es el 8). Sin embargo, la cifra siempre tenderá a la deseada en el peor de los casos, y siempre será igual o menor a la deseada en los casos de conteo completado (es decir, cuando el último paquete reconocido es múltiplo de la ventana). Por el resto, las pruebas concluyeron que el algoritmo funcionaba tal y como estaba planificado.

También se hicieron pruebas de concepto sobre otras características importantes del protocolo en C#. Estas son las conclusiones a las que se llegaron:

- La creación de dos canales de comunicación implica la necesidad de una doble inicialización de la conexión (como puede verse en 3.2.2.1 *HandShake*).
- Para el cálculo del ping era necesario un mecanismo propio, ya que algunos *routers* no contestan a los mensajes ICMP (incluso usando al propia clase con la que cuenta C# para el manejo de pings[22]).
- La máquina que actúe de servidor deberá tener los puertos de ambos canales abiertos para que el cliente pueda conectarse a él. Otra opción es crear una red privada entre las máquinas participantes en la conexión. Para ello pueden usarse softwares como *Hamachi*.
- Para la utilización de sockets, será preferible utilizar la clase *Socket* incorporada en C#. También cuenta con una clase *UdpClient*, pero es una abstracción de más alto nivel.[22]

Para pruebas y resultados más específicos referentes a la implementación de MUSE-RP, consultar "Implementación de protocolos RUDP en el desarrollo de videojuegos multijugador"[21] .

5. Conclusiones y trabajo futuro

En este trabajo se ha diseñado, tras una investigación profunda sobre los algoritmos RUDP y su situación actual en el área de los videojuegos, un protocolo fiable sobre RUDP que cumple todos los objetivos previstos en 3.2. Características deseadas.

El objetivo principal, crear una capa fiable sobre UDP, ha sido posible gracias a la incorporación de mecanismos propios de TCP sobre capa de aplicación , como son los algoritmos de ventana deslizante. De hecho, se ha querido ir más allá, y se le ha dotado al protocolo MUSE-RP de la posibilidad de usar dos de estos algoritmos: uno totalmente fiable y otro parcialmente fiable. Esto ayudará a los desarrolladores a poder elegir qué canal usar según el tipo de mensaje a enviar. Por ejemplo, para el flujo de información de movimiento podrá usarse el canal parcial, mientras que para el flujo de eventos importantes podrá usarse el canal fiable. Además, el porcentaje del canal parcialmente fiable puede configurarse. También destacar que al usar un puerto por cada canal, el tráfico de cada uno se manejará de manera paralela.

Aparte de esta característica principal, se han conseguido incorporar con éxito otro tipo de herramientas, como pueden ser la incorporación de mecanismos de conexión, que facilitan el manejo de la comunicación entre los puntos terminales implicados. Además, dichas conexiones podrán cerrarse en caso de *timeout* o de un mensaje de *End*.

El uso de cálculo de RTT y ping incorporados, aparte de activar los eventos de *timeout* cuando es necesario, también son utilizados para definir el temporizador que usarán los canales de la conexión, adaptándose al estado del canal de una forma sencilla, pero eficaz. Cabe destacar, además, que la incorporación de un ping dentro del protocolo evita posibles problemas en caso de que los puntos terminales no contesten a mensajes ICMP.

Otra herramienta que le será útil a los desarrolladores que usen MUSE-RP es el uso de manejadores por tipo de mensaje, mecanismo que ya se ha comprobado que incorporan varios protocolos. Esto facilita el desarrollo del videojuego, ya que el programador solo tendrá que añadir el método que desee ejecutar cuando se reciba un mensaje de un tipo determinado.

Mencionar, por último, otras características de interés, tales como: el uso de una cabecera pequeña que no sobrecargue el tamaño del mensaje; un enfoque a una arquitectura cliente-servidor, que es la preferida por los desarrolladores; y la

incorporación de mecanismos de TCP que ayudan a reducir la latencia, tales como la retrasmisión rápida.

En definitiva, se ha conseguido un protocolo RUDP altamente configurable y prometedor, que puede llegar a ser una herramienta útil para los desarrolladores de videojuegos que busquen un protocolo sobre UDP de código abierto, y que puedan adaptar a sus necesidades.

En cuanto a posibles mejoras y trabajo futuro de este diseño de MUSE-RP, destacar varias funcionalidades que, por falta de tiempo, no pudieron añadirse:

- Incorporación de un mecanismo de prioridad de mensajes dentro de ambos canales, que priorice el procesado de ciertos mensajes por encima de otros.
- Implementación de herramientas útiles para la creación de salas y partidas, facilitándole al desarrollador la implementación de este tipo de mecanismos.
- Más énfasis en protocolos de reconexión en caso de desconexión involuntaria.
- Creación de clases encargadas de gestionar las estadísticas de conexión, como, por ejemplo, el porcentaje de pérdidas.
- Incorporación de mecanismos de cifrado de mensajes que aumente la seguridad de la comunicación.

Como conclusión final, es necesario destacar el alto potencial de estos algoritmos RUDP dentro del área del desarrollo de videojuegos. Es un campo que, a primera vista, parece estar en fases tempranas de su evolución, pero que abre las puertas a que los desarrolladores puedan crear un protocolo que se ajuste a las necesidades de sus productos.

6. Bibliografía

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

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

[3] 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.

[4]https://www.streamingmediaglobal.com/Articles/Editorial/Featured-Articles/Reliable-UDP-(RUDP)-The-Next-Big-Streaming-Protocol-86388.aspx [Consultado el 29-06-2022]

[5] https://docs.tibco.com/pub/ftl/4.3.0/doc/html/GUID-1F5EE6A1-27D0-4D23-AC98-FFFBC712F747.html[Consultado el 20-06-2022]

[6]https://ims.improbable.io/insights/kcp-a-new-low-latency-secure-network-stack [Consultado el 06-06-2022]

[7] Fang, J., & Liu, M. (2011, September). Design and Implementation of Embedded RUDP. In *2011 Second International Conference on Networking and Distributed Computing* (pp. 7-9). IEEE.

[8] https://github.com/MidLevel/Ruffles [Consultado el 01-07-2022]

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

[10] http://enet.bespin.org/Features.html[Consultado el 20-06-2022]

[11]https://www.ibm.com/products/aspera [Consultado el 30-05-2022]

[12] Thammadi, A. (2011). Reliable user datagram protocol (RUDP).

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

[14]https://www.gamesparks.com/blog/tips-for-writing-a-highly-scalable-server-authoritative-game-part-1/[Consultado el 20-06-2022]

[15] Chen, T. T. (2015). Online games: Research perspective and framework. *Computers in Entertainment (CIE)*, *12*(1), 1-26.

[16] https://github.com/MatthiWare/RUDP.Net[Consultado el 01-07-2022]

[17] https://github.com/fexolm/GServer[Consultado el 01-07-2022]

[18] https://github.com/skywind3000/kcp/blob/master/README.en.md[Consultado el 28-06-2022]

[19] https://mirror-networking.gitbook.io/docs/transports/kcp-transport[Consultado el 20-06-2022]

[20] https://repositorio.uca.edu.ar/bitstream/123456789/522/1/metodologias-desarrollo-software.pdf[Consultado el 28-06-2022]

[21] Martín García, Celtia. (2022). Implementación de protocolos RUDP en el desarrollo de videojuegos multijugador.

[22] https://docs.microsoft.com/es-es/documentation/[Consultado el 22-06-2022]

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

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

7. Anexo: Resultados de la encuesta a desarrolladores junior

7.1. Preguntas Generales

Como es costumbre en este tipo de cuestionarios, primero se ha preguntado sobre datos del encuestado (edad, género(Ilustración 17), perfil profesional y qué carrera se ha estudiado). En total se consiguieron 36 respuestas, y los siguientes resultados:

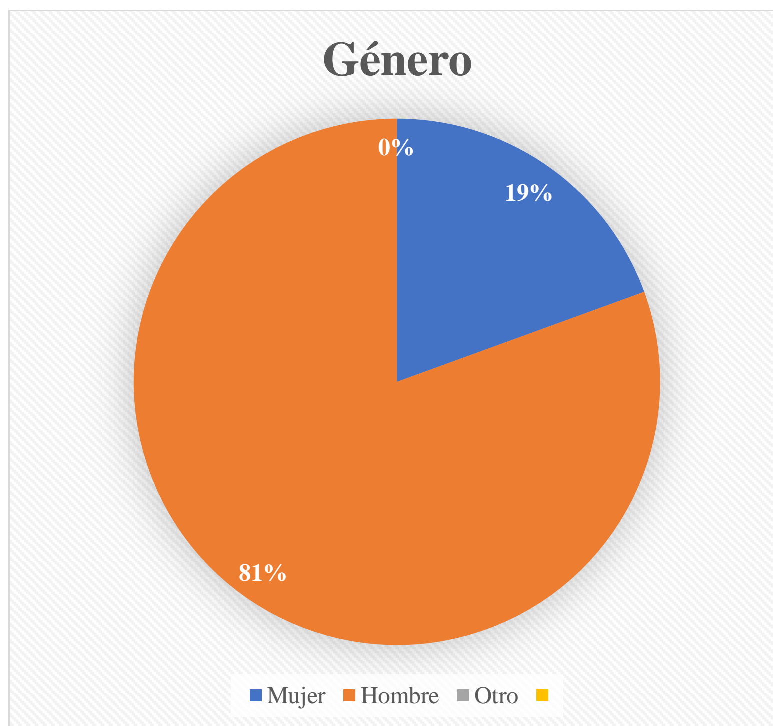

Ilustración 17: Resultados sobre el género de los encuestados

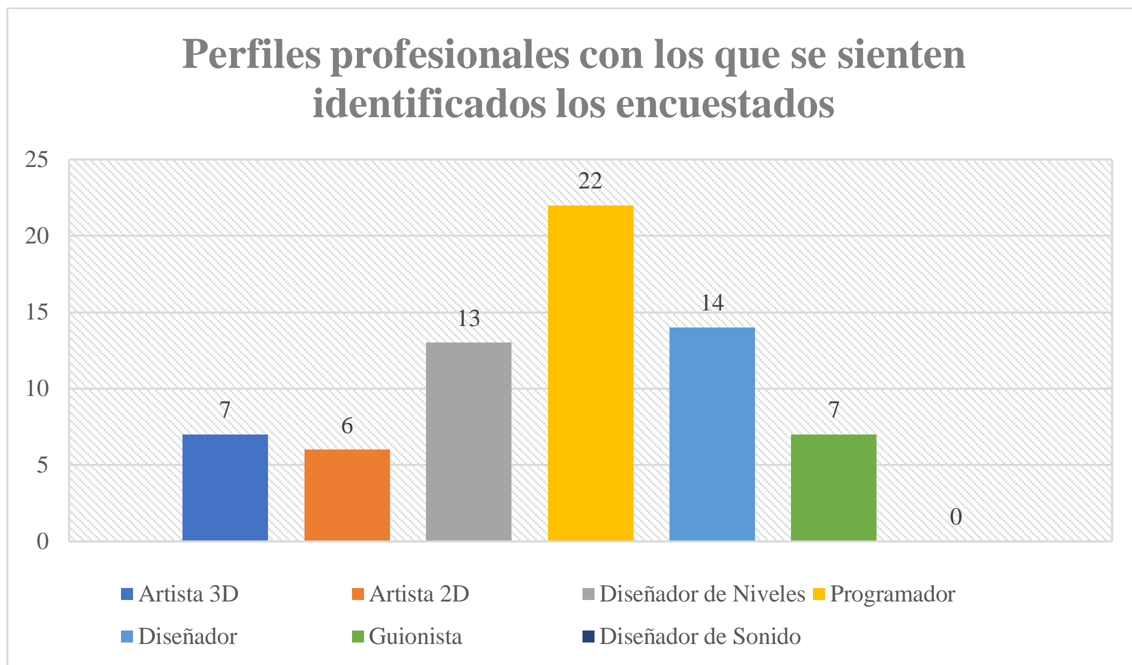

Ilustración 18: Resultados sobre los perfiles profesionales de los encuestados

Como se puede observar en la Ilustración 18, el 61,1% de los encuestados se identifican como programadores (aparte de otros roles). Esto significa que sus respuestas serán muy valiosas, ya que están más familiarizados con el terreno. Aun así, se ha querido conocer también la opinión de miembros de otras áreas del sector, ya que se han querido abordar todos los puntos de vista.

7.2. Preguntas específicas

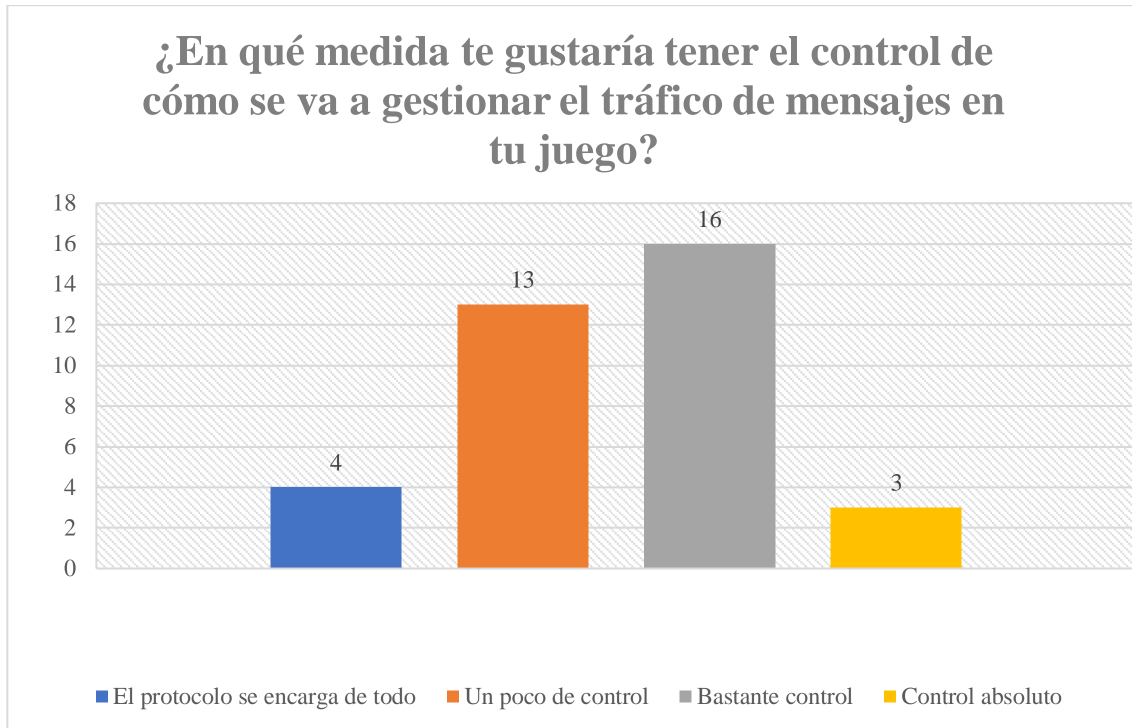

Ilustración 19: Resultados sobre la primera pregunta de la encuesta

Tras estas respuestas resumidas en la Ilustración 19, se puede adivinar que los desarrolladores quieren, por una parte, un protocolo cómodo (que no haya que configurar enteramente), pero a la vez quieren tener un mínimo de control para poder adaptarlo a sus necesidades. Por ello, se han decantado, sobre todo (un 77,78%) por respuestas intermedias.

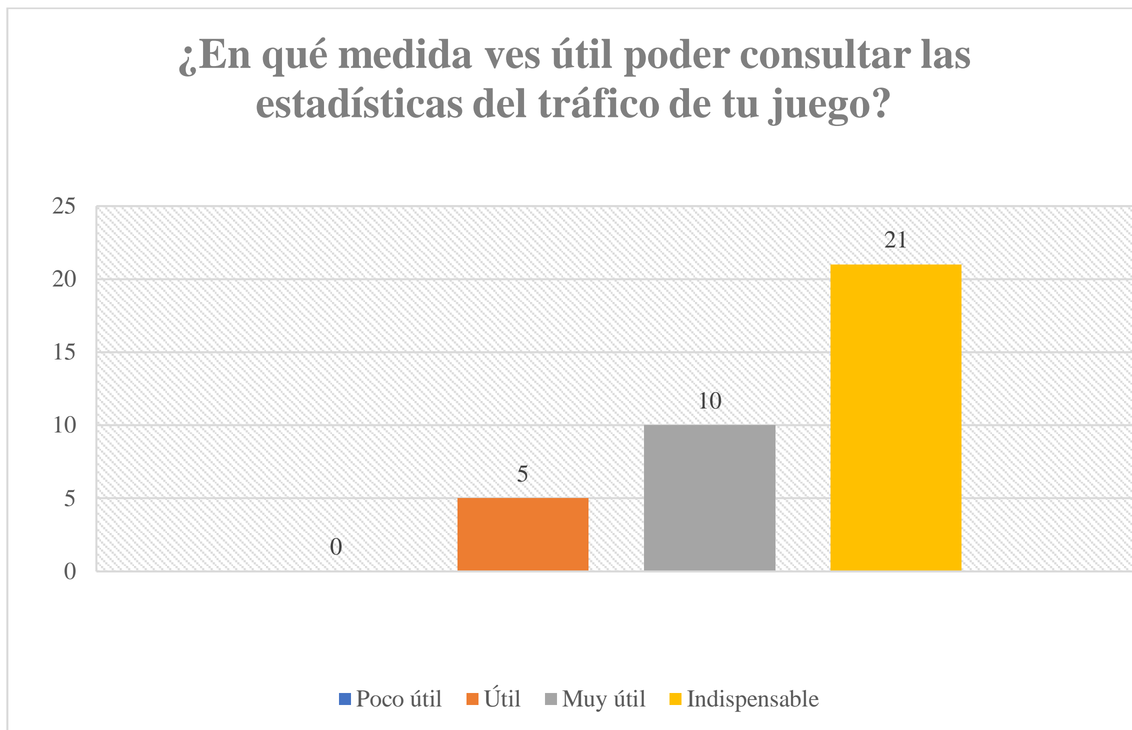

Ilustración 20: Resultados sobre la segunda pregunta de la encuesta

En este caso, más de la mitad de los encuestados (el 58,3%) ve indispensable poder tener acceso a las estadísticas del tráfico, como puede verse en la Ilustración 20. Esto es lógico, ya que tener acceso a ese tipo de información puede ayudar en la toma de decisiones , tanto de los desarrolladores, como de los algoritmos internamente.

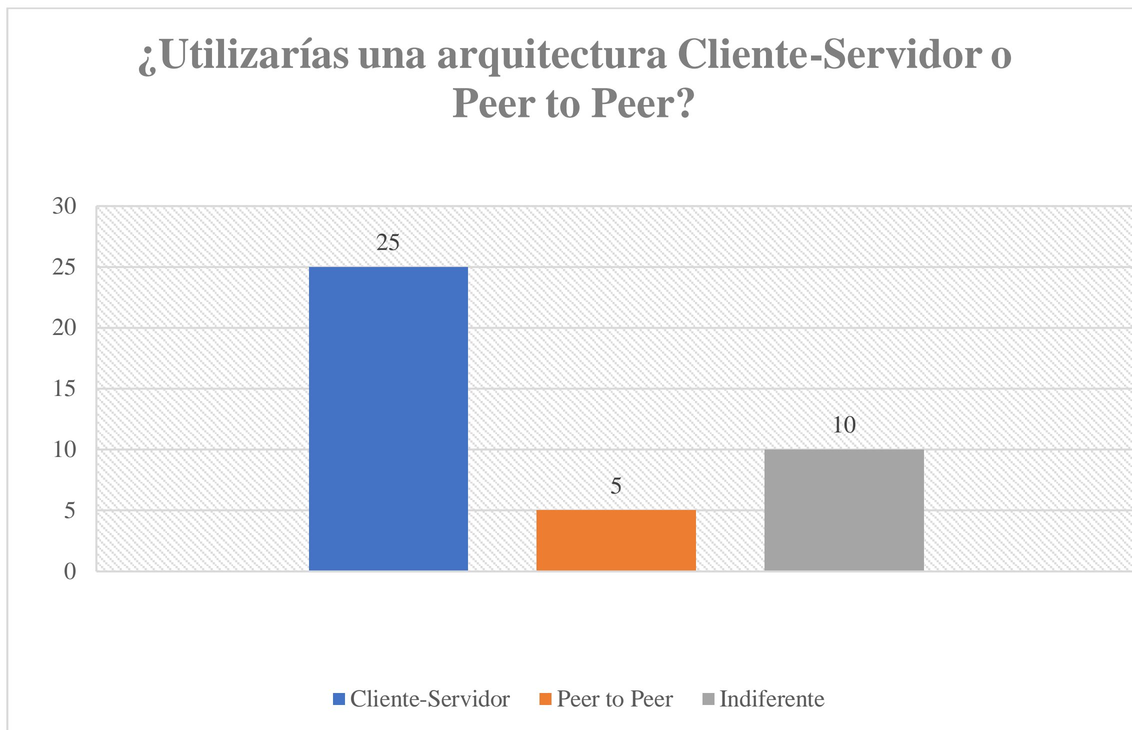

Ilustración 21: Resultados sobre la tercera pregunta de la encuesta

Como se ve en la Ilustración 21,el 69,4% opina que la arquitectura que utilizarían sería un Cliente-Servidor. Quizás sea por su simpleza o porque es con lo que más se está familiarizado. Aun así, 5 personas prefieren una arquitectura *Peer to Peer*, mientras que para 10 de los encuestados la elección les era indiferente.

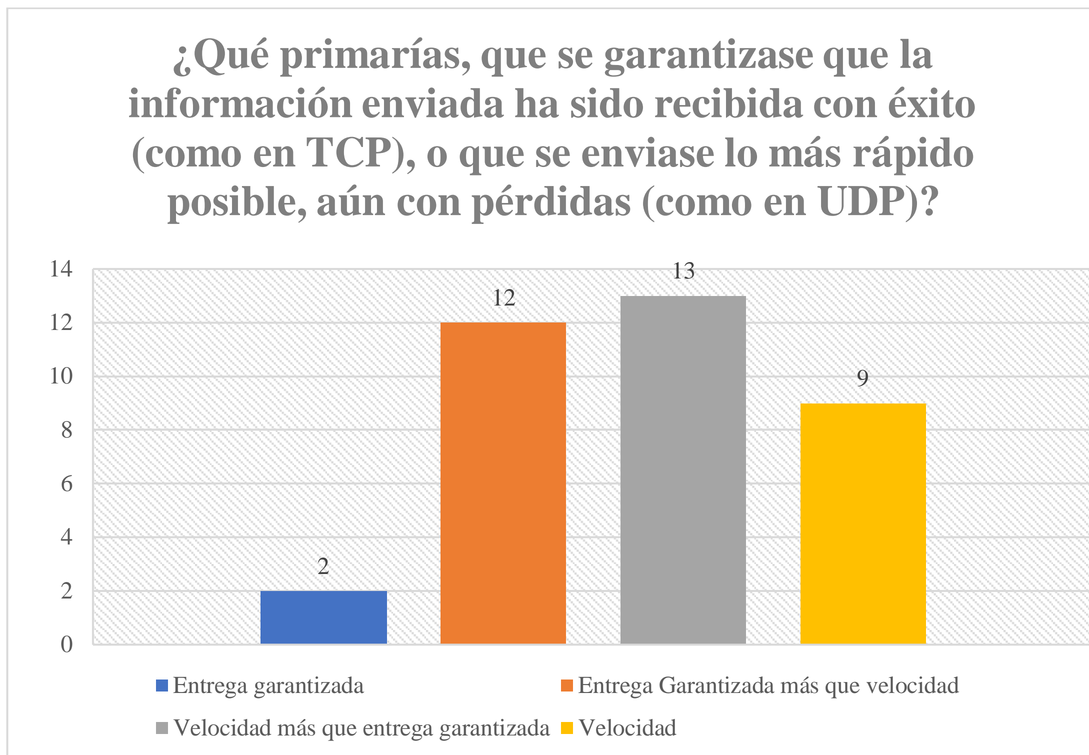

Ilustración 22: Resultados sobre la cuarta pregunta de la encuesta

Como ocurría en la primera cuestión, más de la mitad (el 69,4%) se ha decantado por una respuesta intermedia (puede verse en la Ilustración 22). Es decir, no quieren renunciar a la entrega garantizada, pero tampoco a la velocidad. Aun así, si hay que decantarse por una de las opciones, 22 personas (el 61,1% de los encuestados) priman la velocidad ante la entrega fiable. No es de extrañar, ya que los videojuegos actuales utilizan mucho UDP por esto mismo. Aun así, a muchos les cuesta renunciar a la entrega fiable.

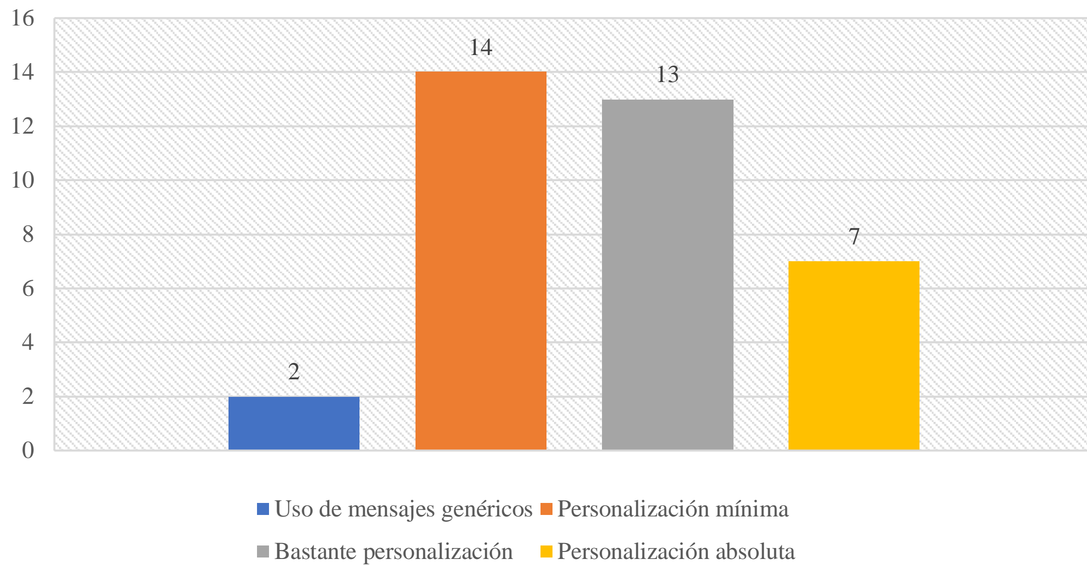

Ilustración 23: Resultados sobre la quinta pregunta de la encuesta

Como se aprecia en la Ilustración 23, esta es otra ocasión en la que la respuesta general es algo intermedio entre la personalización absoluta y la comodidad de los mensajes genéricos (75%). Sin embargo, hay una clara inclinación a la personalización, aunque sea mínima.

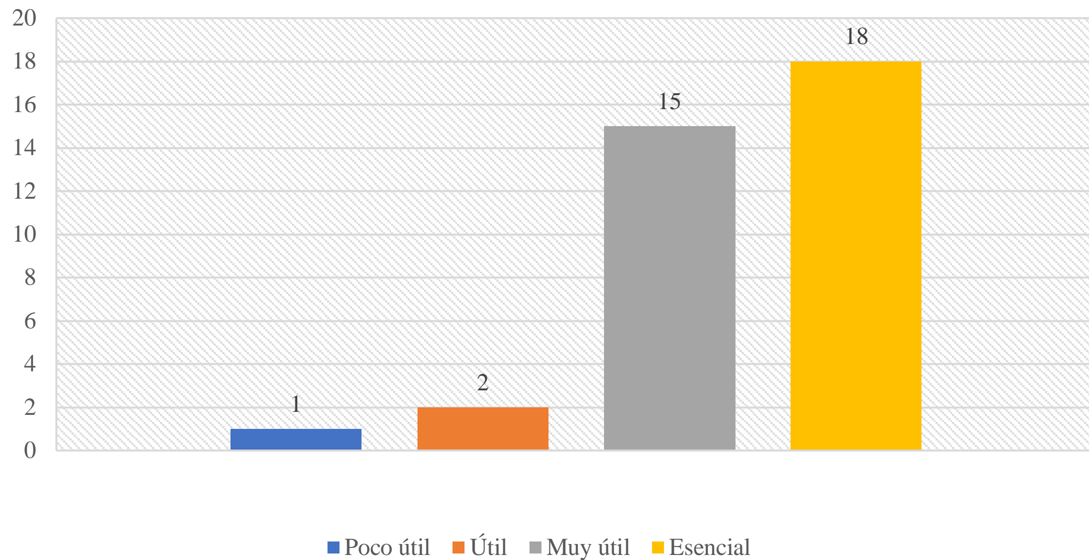

Ilustración 24: Resultados sobre la sexta pregunta de la encuesta

Observando la Ilustración 24, puede verse que la mitad de los encuestados ve esencial poder decidir entre varios modos de envío de mensajes (por ejemplo: un modo fiable con entrega garantizada en todos los casos, un modo poco fiable, con entrega garantizada para un porcentaje específico de mensajes, un modo sin entrega garantizada, etc). De hecho, tan solo el 8% cree que puede no ser muy útil.

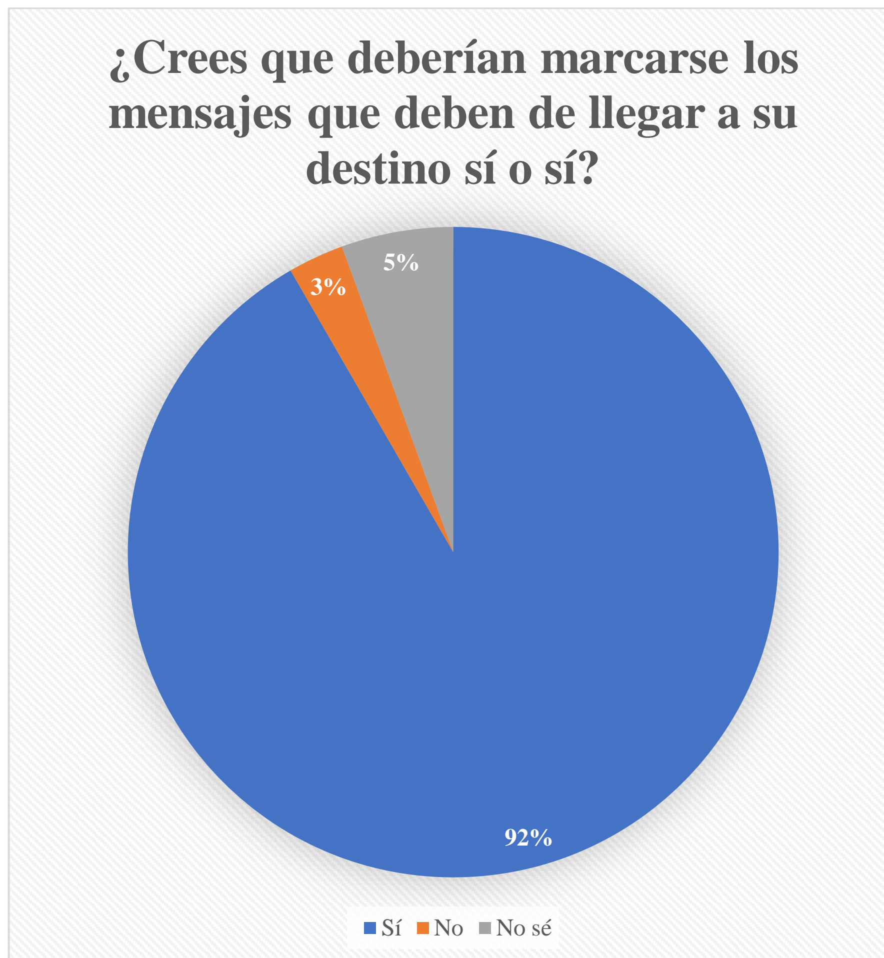

Ilustración 25: Resultados sobre la séptima pregunta de la encuesta

Como se ve en la Ilustración 25, el 91,7% de los encuestados está de acuerdo en que los mensajes que tengan que llegar a su destino deben de señalarse de alguna manera dentro del tráfico de información.

En cuanto a la última pregunta de la encuesta: “¿hay alguna característica específica que quisieras añadir al protocolo que se fuera a utilizar para controlar el tráfico online?”, solo obtuvo una respuesta, ya que era opcional. El encuestado que respondió destacó que la incorporación de mecanismos adicionales de cifrado podría ser muy útil a la hora de aumentar la seguridad del protocolo.

7.3. Conclusiones de la encuesta

Tras analizar las necesidades de los desarrolladores, se pueden deducir varios puntos sobre el protocolo que considerarían ideal para el desarrollo de sus videojuegos:

- Se quiere tener un mínimo de control sobre el tráfico del juego, pero se debe dotar al protocolo de cierta independencia e inteligencia.
- Las estadísticas del envío de información serían útiles para la toma de decisiones.
- La arquitectura Cliente-Servidor es con la que más cómoda se sienten los desarrolladores encuestados.
- La velocidad es esencial para el tráfico online de un videojuego, pero no se puede dejar de lado la entrega fiable: ha de conseguirse un punto medio.
- Sería útil poder personalizar mínimamente los mensajes con la información del juego, así como marcar aquellos que no se puedan extraviar bajo ningún concepto.

8. Anexo: Código en Python de prueba del algoritmo parcialmente fiable

```
Programa principal

ventana, porcentaje, maxFallos= InitialData()
cadena = "Ventana: "+str(ventana) + " Porcentaje: " + str(porcentaje) + " Máximo de fallos
tolerables: " + str(maxFallos)
print(cadena)
Receptor(ventana,maxFallos)
```

Algoritmo 5: Programa principal de la prueba en Python del algoritmo parcialmente fiable.

```
Initial Data

def InitialData():
    entrada= input().split()
    ventana= int(entrada[0])
    porcentaje= float(entrada[1])
    maxFallos= int((porcentaje/100)*ventana)
    return ventana, porcentaje, maxFallos
```

Algoritmo 6: Código de recogida de datos de entrada de la prueba en Python del algoritmo parcialmente fiable.

```
Coger del buffer

def GetBuffer(buffer):
    buffer.sort(reverse=True)
    res= buffer.pop()
    buffer.sort()
    return buffer, res
```

Algoritmo 7: Algoritmo para coger paquetes del buffer de la prueba en Python del algoritmo parcialmente fiable.

```python
def AñadirBuffer(buffer,paquete,ventana,ack):
    if paquete<=ack+ventana: #Es decir, el paquete esta dentro de la ventana de recepcion
        print("\nSe guarda "+ str(paquete)+" en buffer")
        buffer.append(paquete)
        buffer.sort()
    return buffer
```

Algoritmo 8: Algoritmo para añadir paquetes al buffer de la prueba en Python del algoritmo parcialmente fiable.

```python
def Receptor(ventana,maxFallos):
    entrada= input()
    paquete= int(entrada)
    conteo=0
    fallos=0
    buffer=[]
    ack=0
    paquetesLlegados=0
    while(paquete>=0):
        ack, conteo,fallos,buffer,paquetesLlegados=
Algoritmo(paquete,ventana,conteo,fallos,maxFallos,buffer,ack,paquetesLlegados)
        acciones= "\nSe devuelve: " + str(ack) +"\nEl conteo esta en: "+ str(conteo) +"\nLos fallos
a:"+str(fallos)+"\nY el buffer: "+ str(buffer)
        print(acciones)
        if ack!= 0:
            porcentajeFallos= ((ack-paquetesLlegados)/ack)*100
            print("\nPorcentaje de pérdidas: "+str(porcentajeFallos))
        entrada = input()
        paquete = int(entrada)
```

Algoritmo 9: Algoritmo que simula al receptor de la prueba en Python del algoritmo parcialmente fiable.

Algoritmo parcialmente fiable

```python
def Algoritmo(paquete,ventana,conteo,fallos,maxFallos,buffer,ack,paquetesLlegados):
    if len(buffer)>0 and paquete>buffer[0]:
        buffer= AñadirBuffer(buffer,paquete,ventana,ack)
        return ack, conteo,fallos,buffer,paquetesLlegados
    D= paquete-(ack+1)
    if D<0:
        print("Se descarta, duplicado")
        return ack,  conteo, fallos, buffer, paquetesLlegados
    if D==0:
        conteo+=1
        paquetesLlegados+=1
        if conteo>=ventana:
            conteo=0
            fallos=0
        ack=paquete
        if len(buffer)>0:
            buffer,newPaquete= GetBuffer(buffer)
            return Algoritmo(newPaquete,ventana,conteo,fallos,maxFallos,buffer,ack,paquetesLlegados)

        else:
            return ack,conteo,fallos,buffer,paquetesLlegados
    if D>0:
        if D+conteo+1>ventana:
            h1= ventana-conteo
            h2= D-h1
            newFallos= fallos+h1
            if newFallos<=maxFallos and h2<= maxFallos:
                paquetesLlegados += 1
                conteo= h2+1
                fallos= h2
                ack= paquete
                if len(buffer) > 0:
                    buffer, newPaquete = GetBuffer(buffer)
                    return Algoritmo(newPaquete, ventana, conteo, fallos, maxFallos, buffer,
ack,paquetesLlegados)
                else:
                    return ack, -1, conteo, fallos, buffer,paquetesLlegados
            else:
                buffer = AñadirBuffer(buffer, paquete, ventana, ack)
                return ack, conteo, fallos, buffer,paquetesLlegados
        else:
            newFallos=D+fallos
            if newFallos > maxFallos:
                buffer = AñadirBuffer(buffer, paquete, ventana, ack)
                return ack,conteo,fallos,buffer,paquetesLlegados
            else:
                fallos=newFallos
                conteo+=D+1
                paquetesLlegados += 1
                if conteo == ventana:
                    conteo = 0
                    fallos = 0
                ack=paquete
                if len(buffer) > 0:
                    buffer, newPaquete = GetBuffer(buffer)
                    return Algoritmo(newPaquete, ventana, conteo, fallos, maxFallos, buffer,
ack,paquetesLlegados)
                else:
                    return ack, conteo, fallos, buffer,paquetesLlegados
```

Algoritmo 10: Implementación del algoritmo parcialmente fiable de la prueba en Python del algoritmo parcialmente fiable.