FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico - File Transfer

Introducción a sistemas distribuidos - Grupo B10 - 2c2023

Nombre	Padrón
Amigo, Nicolas	105832
Bravo, Nicolas Francisco	106753
Cuppari, Franco	104098
Ledesma, Dylan	102876

1. Introducción

En este Trabajo Práctico se desarrolló un protocolo de Transferencia Confiable de Datos. Siguiendo una estructura Cliente-Servidor, una aplicación de red que facilita la transferencia de archivos mediante las funciones de Subir (Upload) y Descargar (Download) fue implementada. Se utilizó el protocolo UDP para el transporte de datos y se implementaron los protocolos de "Stop & Wait" y "Selective Repeat" con el objetivo de garantizar una transferencia de datos segura y confiable.

2. Hipótesis y suposiciones realizadas

- Dada la libertad para elegir el timeout para un segmento, se eligió como timeout ***2 milésimas*** para los paquetes enviados. Pasada esa cantidad de segundos se asume al paquete como perdido.

- Si no hubo respuesta luego de 10 timeouts en el socket, asumimos que el otro interlocutor de la conexión se desconectó.

- Se elige el tipo de algoritmo de transferencia confiable (Stop & Wait, Selected Repeat) antes de ejecutar la aplicación.

- El tamaño máximo de un archivo para enviar es de 100GB. Se dedican 20 bytes para representar el tamaño del archivo.

- Si se desea cargar en el servidor un archivo que ya existe entonces este es sobreescrito.

- Los archivos que sean descargados por el cliente, cuya transferencia no sea satisfactoria, se eliminarán, del mismo modo ocurre con los archivos que el cliente desee subir al servidor, cuya transferencia no sea satisfactoria.

- La cantidad de clientes máxima conectada al servidor está supeditada a la cantidad de hilos que pueda manejar la CPU.

3. Implementación

Los mensajes pueden ser de cuatro tipos, ***Start***, ***Data, Error y FIN.*** Cada uno de estos es una clase de Python que implementa la interfaz ***Message***. El flujo de mensajes comienza con el cliente enviando una solicitud de handshake, donde envía un número de secuencia, un byte indicando la operación (UPLOAD o DOWNLOAD), el nombre del archivo, el tamaño del mismo, y un byte boolean que indica si se desea usar el protocolo de Stop and Wait (False) o Selective Repeat (True), para todo este mensaje se dedicaron cuarenta y siete bytes, con el detalle del tamaño de cada campo ubicado en el archivo Protocol.py. Una vez que envía esta información correctamente, el servidor responde con un ACK y comienza la transferencia de paquetes de datos relacionados con el archivo en cuestión. En caso de falla, si es porque supera el máximo de tamaño permitido, el cliente envía un byte de ERROR, y se suspende la transferencia, lo mismo ocurre si se desea hacer una descarga que el servidor no tiene en su directorio de storage.

3.1.1. Stop and Wait

Para el protocolo Stop and Wait, se implementó un funcionamiento clásico de Stop and Wait, que consiste en enviar un segmento y luego esperar una confirmación (ACK) del receptor. Si el emisor no recibe el ACK dentro de un tiempo determinado, que es una constante definida en el archivo constants.py y como atributo de la clase Stop and Wait., asume que el paquete se perdió o está dañado, por lo que lo vuelve a enviar. Este enfoque asegura que los datos se entreguen correctamente y en orden, aunque puede causar una baja eficiencia en redes con altas latencias, ya que el emisor debe esperar por cada ACK antes de enviar el siguiente paquete.

La función upload_file es responsable de enviar un archivo al servidor utilizando el método Stop and Wait. Divide el archivo en paquetes de datos y los envía uno por uno al servidor. Si no se recibe el ACK correspondiente después de un tiempo determinado (timeout), el protocolo reenvía el mismo paquete. Se ha implementado un mecanismo para gestionar múltiples timeouts y garantizar la entrega segura de los datos.

La función download_file se encarga de recibir un archivo del servidor usando el protocolo Stop and Wait. Espera la recepción de paquetes de datos, envía ACKs correspondientes y guarda los datos en un archivo local. Se implementa un manejo de timeouts similar al del upload para manejar posibles errores de transmisión.

La clase cuenta con funciones auxiliares para el envío de paquetes, la espera de ACKs y la gestión de timeouts. También se incluye un mecanismo para manejar el cierre de la conexión en caso de errores persistentes.

El protocolo implementa un mecanismo robusto para manejar timeouts y otros posibles errores de transmisión. Cuando se alcanza un número máximo de timeouts (configurable), el protocolo finaliza la conexión para evitar una espera indefinida y garantizar la eficiencia en el caso de fallos de red.

A continuación, se adjuntan representaciones gráficas de distintos casos de ejecución del protocolo Stop and Wait en distintos contextos:

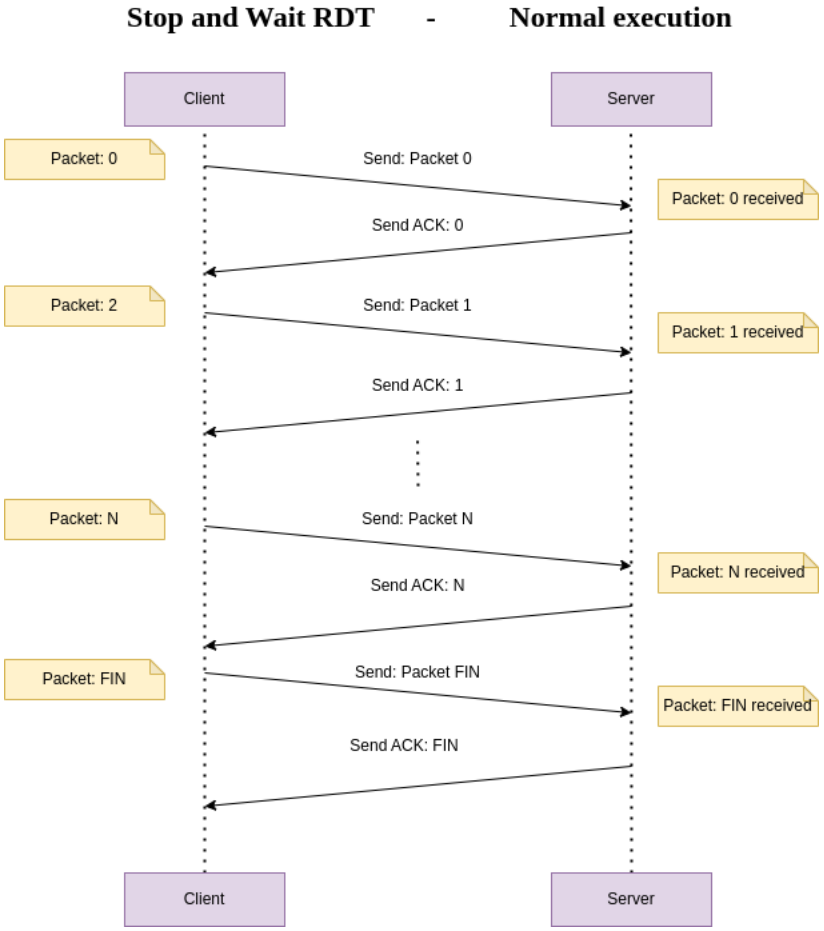


Figura 3.1 Representación gráfica del flujo de mensajes en un contexto normal con el protocolo Stop and Wait implementado.

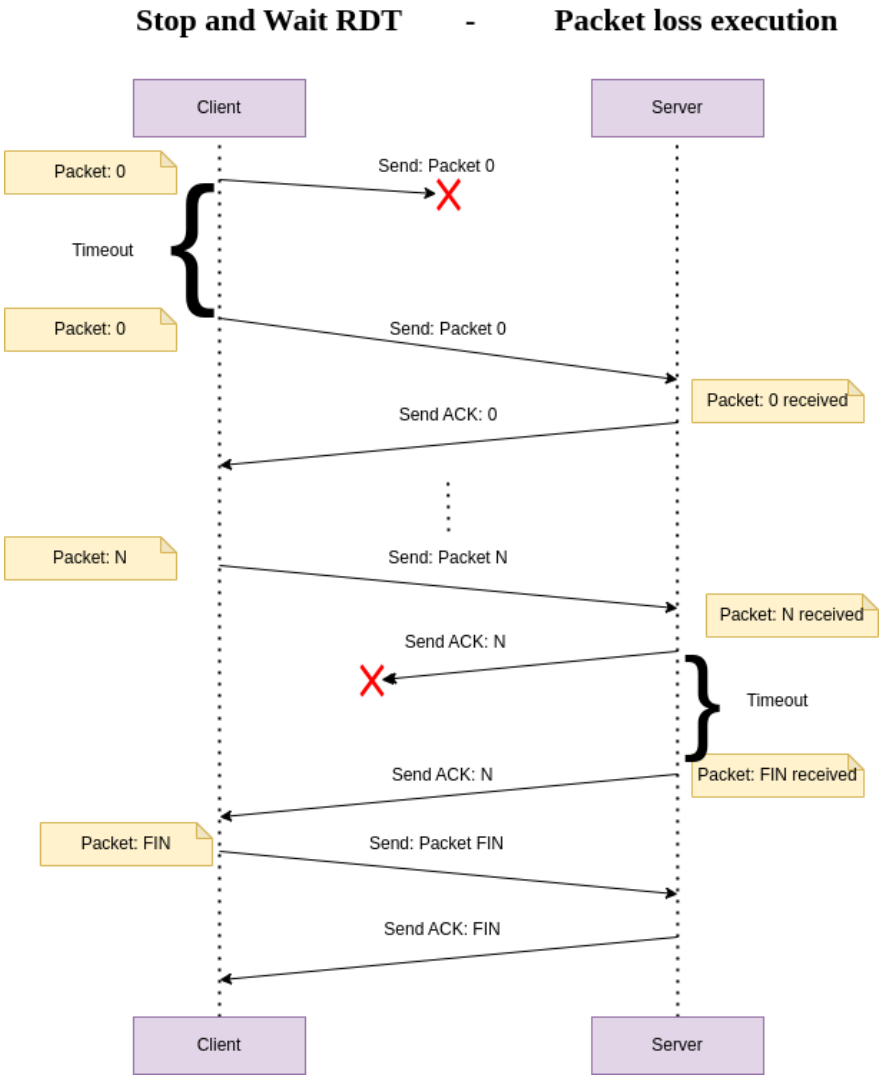


Figura 3.1 Representación gráfica del flujo de mensajes en un contexto de pérdida de paquetes con el protocolo Stop and Wait implementado.

3.1.2. Selective Repeat

Para el protocolo Selective Repeat, se siguió la definición del algoritmo especificada por la bibliografía otorgada, con algunos detalles a continuación.

La clase SRPacket representa un paquete individual en el protocolo Selective Repeat. Cada instancia de esta clase contiene información sobre el estado del paquete (no enviado, esperando ACK o confirmado) y los datos a transmitir, el número de secuencia se encuentra definido por su posición en el array de paquetes. Además, se implementaron mecanismos de timeout para gestionar situaciones donde los paquetes no son confirmados en un tiempo determinado, con este tiempo siendo configurable.

La clase SelectiveRepeat coordina la transmisión y recepción de paquetes mediante el protocolo mencionado anteriormente. Esta utiliza una ventana deslizante para enviar múltiples paquetes antes de recibir confirmación, optimizando así el flujo de datos. Los paquetes no confirmados son retransmitidos selectivamente, minimizando la retransmisión innecesaria y mejorando la eficiencia de la transmisión.

La función upload_file se encarga de enviar un archivo implementando el protocolo selective repeat, esto lo logra siguiendo los siguiente pasos:

- Si hay paquetes disponibles en la ventana, se guardan los siguientes bytes leídos desde el archivo, y se marca el paquete para enviar.

- Se envían los paquetes marcados para enviar en la ventana actual.

- Se esperan los ACK y se actualiza la base del array según corresponda.

- Se evalúan los timers de los paquetes previamente enviados, se envían nuevamente aquellos paquetes en timeout.

La función download_file se encarga de recibir un archivo implementado el protocolo selective repeat, para ello sigue los siguientes pasos:

- Si intenta recibir una ventana de paquetes, sobre aquellos recibidos con un número de secuencia correspondiente a la venta actual, se envía el ACK.
- Si hay nuevos paquetes recibidos, estos se escriben en el archivo, siempre y cuando cumplan con el orden indicado por el número de secuencia.
- Se actualiza la base, tomando el valor del último paquete recibido en el orden correspondiente.

Particularmente, esta implementación dispone de:
- un tamaño de ventana : 20 paquetes
- un tamaño del vector de paquetes: 50 paquetes

En el siguiente diagrama consideramos que el cliente realiza la subida del archivo, y el servidor es quien lo recibe, en él podemos observar como la ventana de paquetes se actualiza en un escenario sin pérdida de paquetes ni ACKs.

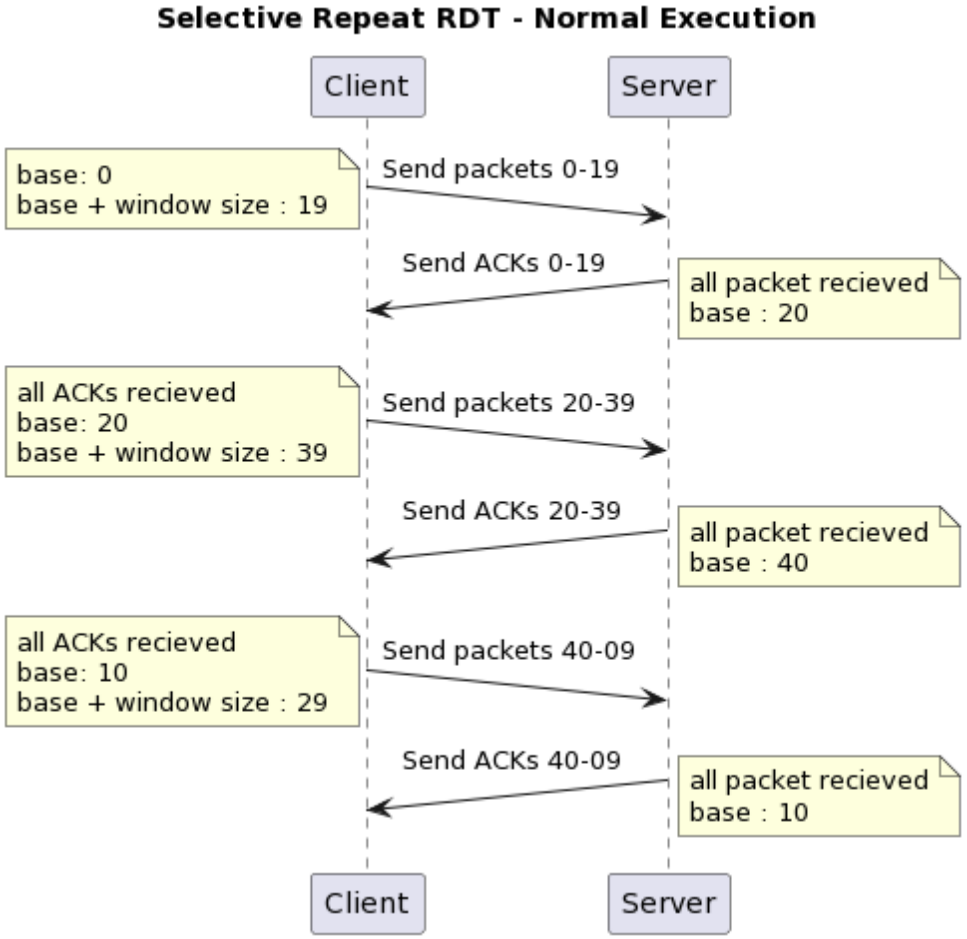


Figura 3.3Representación gráfica del protocolo Stop and Wait en un contexto de ejecución sin pérdida.

Observamos en detalle cómo se comporta frente a la pérdida de paquetes, el protocolo debe seguir intentando enviar nuevos paquetes mientras lo hayan disponibles en la ventana, si existe la pérdida de paquetes la base dejara de actualizarse y la ventana se volverá estática, por esto, luego de cada intento de recibir ACKs, se evalúan los timers.

Eventualmente, en caso de pérdida los timers informaran un timeout y el paquete se enviará nuevamente, una vez que se reciban y confirmen los paquetes perdidos la base se actualizará y la ventana tendrá nuevos paquetes disponibles para enviar.

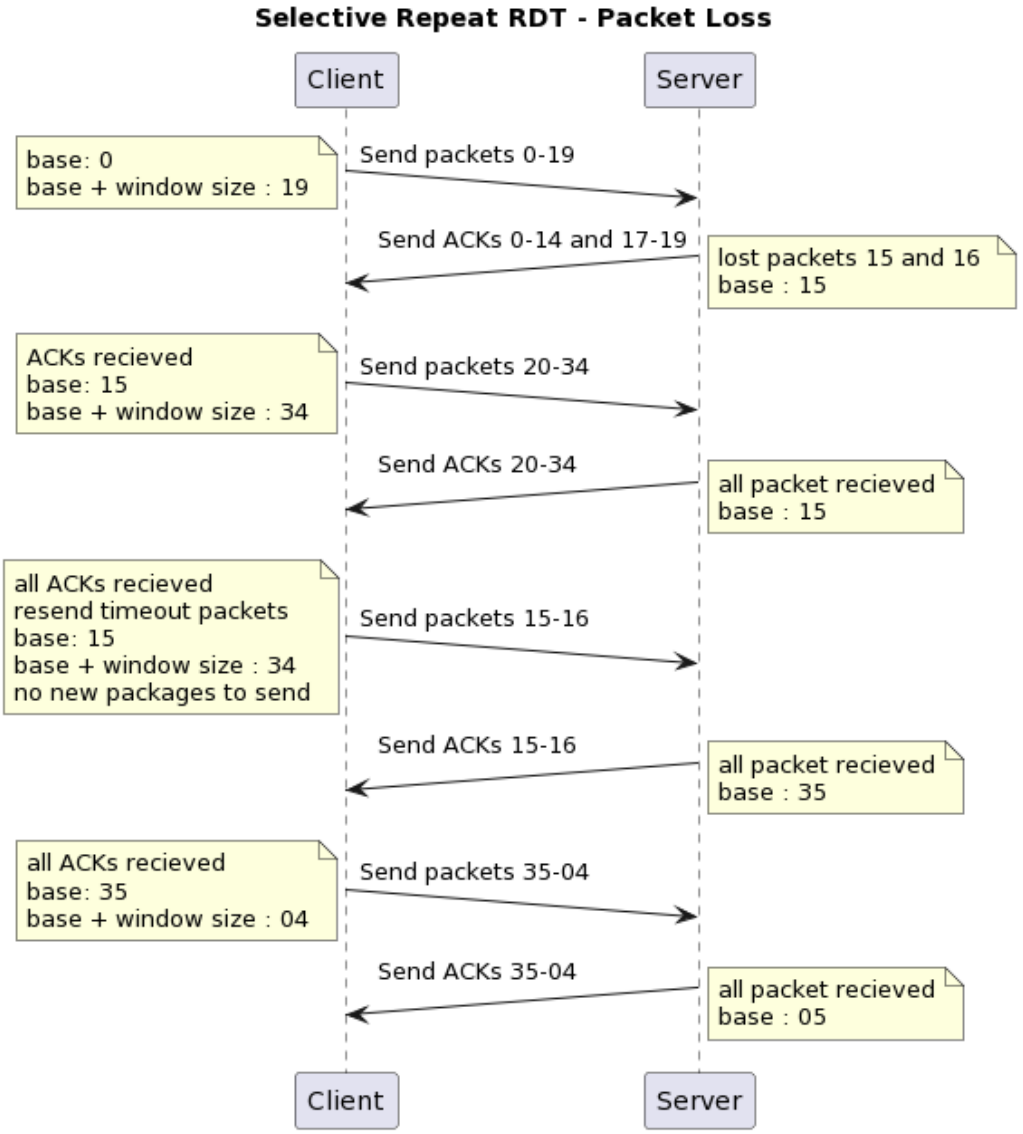


Figura 3.4 Representación gráfica del protocolo Stop and Wait en un contexto de ejecución con pérdida.

Únicamente una vez que todos los bytes hayan sido transferidos y confirmados, el nodo que está corriendo la función upload_file va a enviar un paquete para informar el final de la transferencia a su contraparte, esto es así para reenviar los paquetes perdidos en caso de que existan, aunque el archivo se haya terminado de leer.

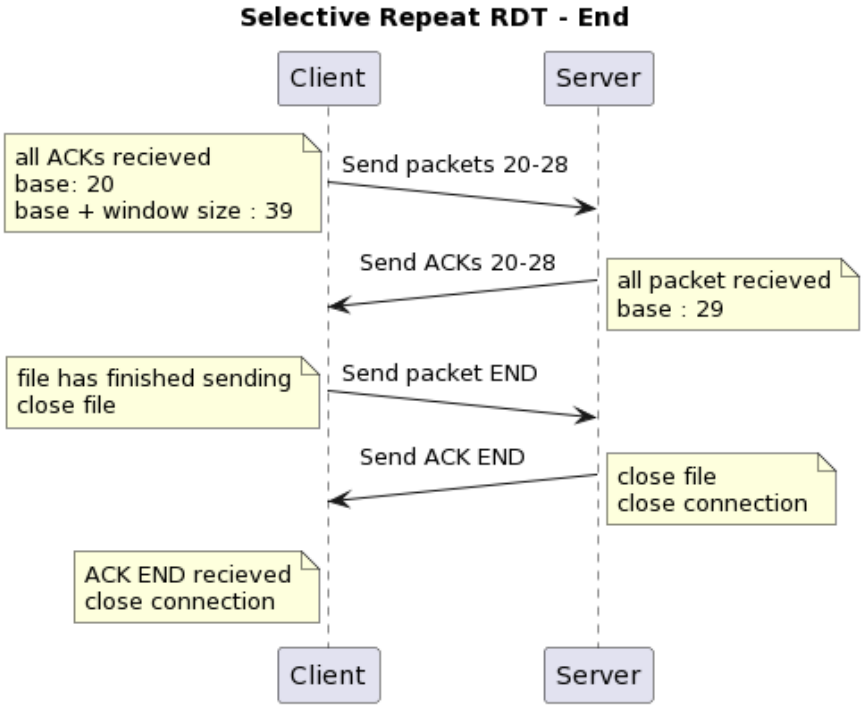


Figura 3.5 Representación gráfica del fin de la ejecución del protocolo Selective Repeat implementado.

4. Pruebas

4.1. Análisis de Performance

Para medir la velocidad de ambos protocolos se hizo la operación de descarga de archivos de varios tamaños con una pérdida del 10% de paquetes. Se midió el tiempo que se tardo con cada protocolo corriendo el comando time de UNIX cinco veces y promediando los resultados.

	Selective Repeat	**Stop and Wait**
100 KB	0.04459714889526367	0.05902814865112305
512 KB	0.012774205207824707	0.039443254470825195
1 MB	0.01591966152191162	0.049703359603881836
2 MB	0.02227621078491211	0.05859255790710449
5 MB	0.2044399929046631	0.19575095176696777

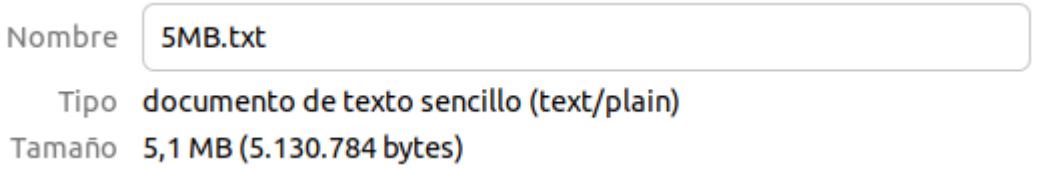
Figura 4.1. Resultados promedio de operaciones de descarga en cada protocolo, indicados en segundos.

4.2 Captura de Paquetes con Wireshark

Una vez que empieza el proceso de handshake, se empiezan a intercambiar paquetes a través de la red. Se hizo, a modo de ejemplo, una prueba en la cual un cliente desea cargar un archivo al servidor, llamado *5MB.txt*. De este modo el primer mensaje será:

000001}}}}}}}}}}}}}}5MB.txt}}}}}}}}}}}}}}51307841

siendo, los primeros cuatro bytes el número de secuencia, el quinto un 0 que representa el tipo de mensaje START, y luego un UNO (True) que indica que el cliente desea subir un archivo al servidor. Para el nombre del archivo se destinan veinte bytes, y como hay nombres más pequeños, se decidió usar un carácter poco habitual en los nombres de archivos, en este caso fue un "}". Por ende, como el nombre del archivo elegido tiene un largo de siete caracteres, hay trece de padding. Los siguientes veinte bytes fueron destinados al tamaño del archivo, se usaron veinte bytes para poder soportar tamaños muy grandes de archivos, para que el protocolo de transporte de datos sea más realista. En este caso, el tamaño exacto del archivo es de 5.130.784 bytes:



Finalmente, el último byte es un booleano que en caso de ser True (como ahora) indica que se utilizará el protocolo Selective Repeat.

A continuación, vemos lo que informa Wireshark:

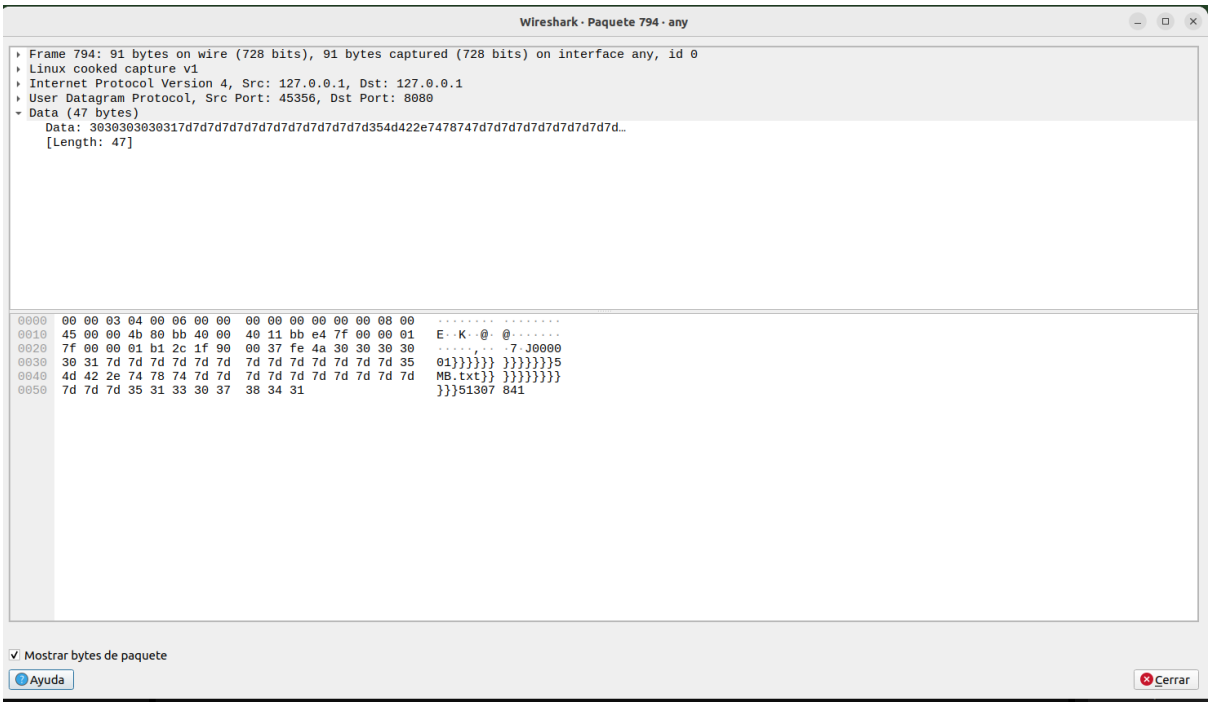


Figura 4.2 Fotografía de la captura de un paquete inicial en Wireshark

Y se observa que Wireshark avala lo explicado anteriormente.

Luego, en respuesta a la información recibida, el Server envía simplemente un byte, con el número tres, que, si se observa en el archivo *constants.py* podemos observar que significa que es un ACK.

A continuación lo indicado por Wireshark:

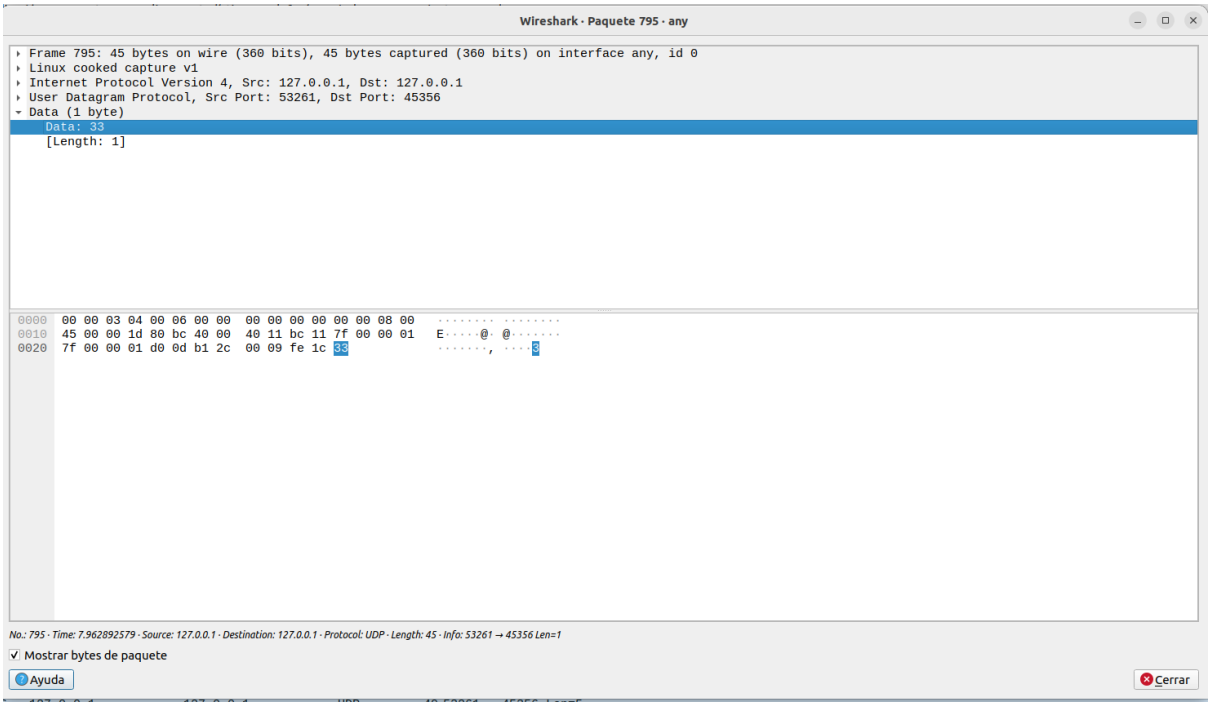


Figura 4.3 Fotografía de la captura de un ACK inicial en Wireshark

Es importante remarcar que estos paquetes son consecutivos, por ende, el orden en que se envían y reciben es correcto.

Otro ejemplo interesante de remarcar en Wireshark es el del envío de datos. El tamaño máximo de paquete de datos es de dieciséis mil trescientos ochenta y ocho bytes, donde los primeros cuatro corresponden al número de secuencia del paquete. Por lo tanto vemos que Wireshark envía:

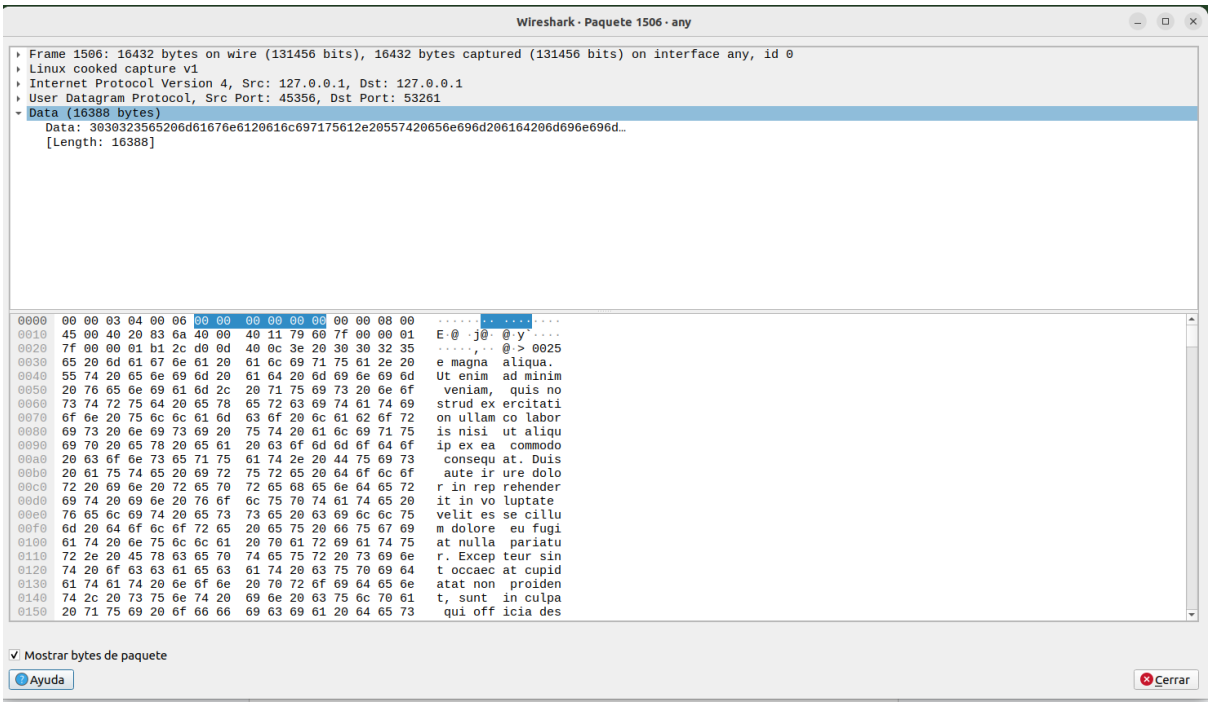


Figura 4.4 Fotografía de la captura de un paquete con número de secuencia y datos en Wireshark.

Donde se ve claramente que los primeros cuatro bytes corresponden al número de secuencia, y los dieciséis mil trescientos ochenta y cuatro bytes son caracteres del archivo que está siendo enviado.

El último ejemplo que resulta interesante de remarcar es el último mensaje, es decir, un byte de FIN, que se representa con el número seis, y tiene el número de secuencia delante. Viendo Wireshark, observamos:

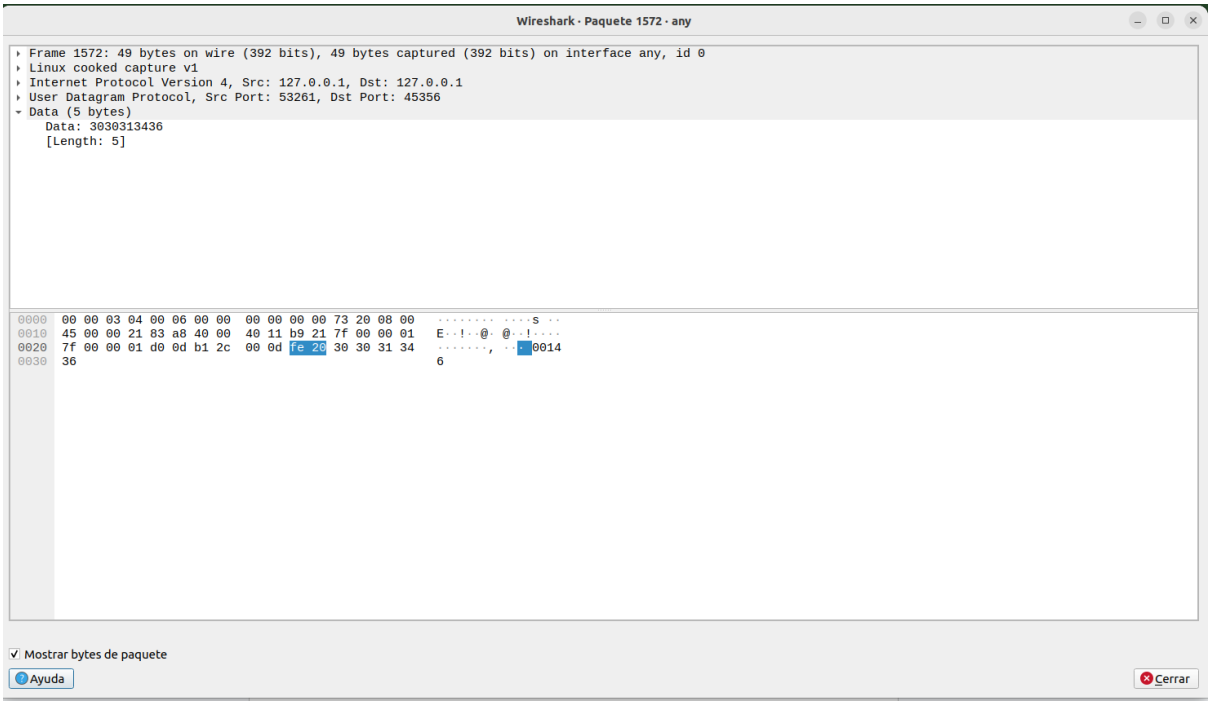


Figura 4.5 Fotografía de la captura de un paquete FIN en Wireshark

Nuevamente, lo que se observa concuerda con lo dicho anteriormente, primeros cuatro bytes de número de secuencia, y el último de FIN (6).

5. Preguntas

1. Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor es un modelo de computación distribuida donde las tareas se dividen entre los sistemas de cliente y servidor, conectados a través de una red. Los clientes, que son las interfaces de usuario, envían solicitudes al servidor para acceder a recursos o servicios específicos. Por otro lado, los servidores gestionan estos recursos, procesan las solicitudes y devuelven los resultados a los clientes.

Los clientes son responsables de la interacción con los usuarios y el envío de solicitudes, mientras que los servidores se encargan de gestionar los recursos, realizar cálculos, acceder a bases de datos y proporcionar servicios específicos. Esta arquitectura permite una distribución eficiente de la carga de trabajo, facilita la administración centralizada de recursos y proporciona una forma escalable de diseñar sistemas que pueden manejar múltiples solicitudes de clientes simultáneamente. En resumen, el cliente solicita y muestra información, mientras que el servidor procesa estas solicitudes y proporciona los datos o servicios solicitados.

2. ¿Cuál es la función de un protocolo de capa de aplicación?

Un protocolo de capa de aplicación establece reglas para que las aplicaciones en dispositivos diferentes se comuniquen en una red. Controla el formato de los datos, la transferencia de información, la seguridad y la gestión de errores. Ejemplos incluyen HTTP para navegación web y SMTP para correo electrónico. Estos protocolos aseguran una comunicación efectiva entre aplicaciones en una red.

3. Detalle el protocolo de aplicación desarrollado en este trabajo.

En este trabajo, se decidió encapsular el trabajo de escribir un archivo en una clase llamada FileReader, que recibe los bytes y los escribe en el archivo correspondiente. De tal manera que una vez que el protocolo elegido para la transferencia por el cliente, se encarga de realizar la tarea de “delivery” entre cliente y servidor, para que el FileReader pueda encargarse de escribir los bytes correctos que serán recibidos de la capa de transporte. Esta clase FileReader, además, se encarga del correcto tratamiento del archivo, ya que se encarga de abrirlo, crearlo si no existe, leerlo, escribirlo, cerrarlo ordenadamente, y eliminarlo si la transferencia no se dió correctamente.

Por otro lado, se crearon los archivos *upload.py* y *download.py* con el objetivo de poner como punto de partida estos archivos del lado del cliente para cada una de las dos operaciones que se deseen realizar. Dentro de este archivo se encuentran los métodos necesarios para el handshake con el Servidor, para tratar de implementar un protocolo orientado a la conexión, pero con sockets UDP.

4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuándo es apropiado utilizar cada uno?

UDP provee el servicio mínimo de envío de paquetes entre dos procesos. No admite conexiones entre hosts ni provee transporte de datos confiable. Es decir únicamente provee el servicio de multiplexación y demultiplexación. Debido a este servicio mínimo UDP es el más rápido de los dos.

UDP es utilizado para servicios que admiten pérdida de paquetes como streaming o llamada de voz que necesitan la mayor velocidad.

TCP, en cambio, provee un transporte orientado a conexiones de datos confiable con garantía de que los paquetes se envíen y el orden sea el correcto.

TCP es utilizado para servicios que requieren la entrega de todos sus paquetes y la velocidad no es una de sus principales prioridades.

6. Dificultades Encontradas

6.1. Dificultades para decidir el tamaño de los paquetes

La primera dificultad que se presentó al momento de comenzar este proyecto fueron una serie de interrogantes como por ejemplo, *¿habrá muchos tipos de mensajes distintos? ¿Estos mensajes deben tener todos el mismo largo? ¿su largo varía, o es fijo?*. Finalmente, luego de días de discusión y debate, la conclusión a la que se llegó, fue que habría cuatro tipos de mensaje, START para el handshake, donde se incluirían el número de secuencia, el tipo de operación, el nombre del archivo, el tamaño del mismo, y el protocolo a través del cual se enviará dicho archivo. El segundo tipo de mensaje es ACK, que consta del número de secuencia del mensaje y un byte que indica que el tipo de mensaje es un ACK. Este último byte puede variar dependiendo de si es un ACK para el mensaje de FIN, o un ACK para el mensaje de DATA. El mensaje de tipo ERROR, también son cinco bytes, con los primeros cuatro perteneciendo al número de secuencia, y el restante indica el tipo de error, como puede ser *FileTooBig, FileNotFound,* entre otros. Y el último tipo de mensaje, *DATA*. Este tipo de mensaje consta de dieciséis mil trescientos ochenta y ocho bytes (aunque la cantidad de bytes es regulable), los primeros cuatro pertenecen al número de secuencia, y los restantes son bytes que luego serán escritos en el archivo en cuestión.

Finalmente, la respuesta a los interrogantes planteados en el párrafo anterior, fue que, los mensajes pueden variar su largo, y no son todos iguales, aunque algunos comparten tamaño en términos de bytes, y entre todos comparten que los primeros cuatro bytes están destinados al número de secuencia.

6.2 Dificultades con la implementación de un handshake apropiado

El siguiente problema fue definir qué cosas habría que incluir en el handshake. Luego de más tiempo de análisis, se concluyó en enviar los campos: número de secuencia, tipo de mensaje, tipo de operación, nombre del archivo, tamaño del archivo y protocolo a utilizar.

Esta decisión permitió captar de forma prematura errores en el servidor a la hora de cargar archivos a este, como por ejemplo, que el archivo sea muy grande, o que este no exista en el directorio donde se almacenan estos.

6.3 Dificultades con la implementación de los protocolos

Estas dificultades surgieron con ambos protocolos, pero el Selective Repeat generó más dificultades porque una vez que se solucionaba un problema, surgían otros dos. Pero algo en común que se notó, fue el hecho del minucioso tratamiento que había que dedicarle al timeout, tanto de la interfaz del socket, como del tiempo que se debía aguardar para volver a envíar un mensaje por el que no se había recibido ACK.

7. Conclusión

Gracias al desarrollo de este proyecto se logró adquirir mayor familiaridad con los protocolos de Transferencia Confiable de Datos y lo que conlleva implementarlos. En específico como asegurar el arribo de cada paquete y que estos se usen en el orden que le corresponde.

Se observó con suma profundidad los protocolos de *Stop and Wait* y *Selective Repeat* y como se componen al momento de querer implementarlos. Además pudimos comparar la velocidad de transferencia entre cada uno con nuestro protocolo y determinar el más rápido.

Finalmente, este proyecto sumado a la teoría dada en clase permitieron adquirir la capacidad de crear un protocolo de capa de transporte propio que corre por sobre UDP. Esto, agregado a un protocolo de capa de aplicación también propio, posibilitó la implementación de un servicio de descarga/subida de archivos.