.UBAfiuba
FACULTAD DE INGENIERÍA

Introducción a los sistemas distribuidos
(75.43) Esteban Carisimo y Juan Ignacio Lopez Pecora

Trabajo Práctico 1
File Transfer

03 de Octubre de 2023

Integrantes:

- Agustina Bocaccio (106393)
- Nicolas Pinto (105064)
- Fernando Balmaceda (105525)
- Valentina Adelsflügel (108201)

Índice

1. Introducción

En este trabajo práctico se implementa un File Transfer haciendo uso del protocolo de transporte UDP (User Datagram Protocol) con ciertas extensiones para que cumpla con las condiciones de un protocolo RDT (Reliable Data Transfer).

Desarrollamos una aplicación de arquitectura cliente-servidor que con las siguientes funciones:

UPLOAD: Transferencia de un archivo del cliente hacia el servidor.

DOWNLOAD: Transferencia de un archivo del servidor hacia el cliente.

Además, posee dos versiones: una con Selective Repeat y otra con Stop and Wait.

2. Hipótesis y suposiciones realizadas

- Los paquetes no vienen corrompidos porque UDP ya cuenta con ese servicio de validación.
- Tanto servidor como cliente, serán corridos en localhost.
- Como la cantidad de bits que puede tomar el número de secuencia es de 32 bits (como se detalla en la sección 3.1), el máximo representable es 2^{32}-1. Esto nos pone una restricción en el tamaño máximo de archivo, dado que si cada paquete contiene 3682 bytes de datos (sin header), sólo podemos enviar archivos que se dividan en 2^{32} paquetes. Esto se traduce en un tamaño de archivo máximo de 2^{32} * 3862 B. Sin embargo, dado que es un número enorme, decidimos tomar como máximo 1GB.

3. Implementación

3.1. Mensajes

La definición del formato de los mensajes fue nuestro punto inicial de análisis. Decidimos que los mensajes tendrán un tamaño fijo, con un header de tamaño predefinido y con la información para la descarga y la subida de los archivos con tamaño variable, lo que permite facilitar el procesamiento de datos del header y mejorar el rendimiento del protocolo.

El tamaño elegido para el mensaje es 4096 bytes (4 kb), ya que de esta forma es lo suficientemente grande para mandar la información necesaria en el header para realizar la comunicación y a la vez enviar una cantidad significativa de datos del archivo que se quiera subir o descargar.

El header tiene un tamaño fijo de 414 bytes y está conformado por los siguientes campos:

- Comando (1 byte) - UPLOAD (1) o DOWNLOAD (2)
- Flags (1 byte) - es un entero que indica una característica del mensaje
  - HI: indica que el mensaje enviado es el primero que manda el cliente para iniciar la conexión con el servidor.
  - CLOSE: indica que el que lo manda quiere terminar la conexión.
  - ACK: indica que el mensaje que se está mandando es un acknowledge del mensaje que recibió la entidad.
  - HI_ACK: indica que el servidor recibió el HI del cliente y está online para establecer la conexión.
  - CLOSE_ACK: indica que recibió el CLOSE y va a cerrar la conexión.
  - ERROR: indica que el mensaje contiene un mensaje de error.
  - LIST: indica que se quiere obtener un listado de los archivos disponibles en el servidor.
  - NO_FLAGS: el mensaje no tiene ninguna característica de las mencionadas anteriormente.
- Data Length (4 bytes): Es el largo en bytes de los datos enviados.

- File name: UTF-8 string (400 bytes), es el nombre del archivo que se quiere descargar o subir.
- Sequence number (4 bytes): es un entero sin signo que indica el número de secuencia del mensaje que se está enviando. Al empezar la comunicación está en 0.
- ACK number (4 bytes): indica hasta cuál segmento se recibió del otro lado de la comunicación.

Los bytes restantes del mensaje están destinados a los datos del archivo que se quiere descargar o subir. Entonces, los bytes destinados a la información del archivo son variables, pero tienen un máximo de 4096 - 414 (3682) bytes.

3.1.1. UPLOAD

El comando UPLOAD habilita a los usuarios para cargar archivos en un servidor con la opción de elegir un nombre personalizado. En caso de que no se seleccione un nombre, se utiliza el nombre predeterminado del archivo cargado.

Este proceso permite cargar archivos con nombres duplicados, ya que el servidor realiza una verificación previa para determinar si ya existe un archivo con el mismo nombre. Si detecta una coincidencia, el servidor agrega automáticamente un número al nombre del nuevo archivo para evitar confusiones con archivos existentes que compartan el mismo nombre. De esta manera, se garantiza la integridad y la organización de los archivos en el servidor.

```
usage: upload [-h] [-v | -q] [-H HOST] [-p PORT] [-n NAME] -s SRC [-pr RDTPROTOCOL]

This is a program to upload files to a server

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         increase output verbosity
  -q, --quiet           decrease output verbosity
  -H HOST, --host HOST  server IP address
  -p PORT, --port PORT  server port
  -n NAME, --name NAME  file name
  -s SRC, --src SRC     source file path
  -pr RDTPROTOCOL, --RDTprotocol RDTPROTOCOL
                        stop_and_wait (sw) or selective_repeat (sr)
```

3.1.2. DOWNLOAD

Con el comando DOWNLOAD, los usuarios tienen la posibilidad de acceder a la descarga de archivos desde el servidor. Esta acción se realiza utilizando el nombre proporcionado al momento de cargar el archivo en el servidor. Además, los clientes pueden especificar la ubicación exacta donde desean almacenar el archivo descargado. En caso de no especificar una ubicación, el archivo se descargará automáticamente en una ubicación predeterminada. Esta flexibilidad y personalización permiten una experiencia de descarga eficiente y adaptada a las necesidades individuales de cada usuario.

```
usage: download [-h] [-v | -q] [-H HOST] [-p PORT] [-n NAME] [-d DST] [-pr
    RDTPROTOCOL] [-f]

This is a program to download files from a server

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         increase output verbosity
  -q, --quiet           decrease output verbosity
  -H HOST, --host HOST  server IP address
  -p PORT, --port PORT  server port
  -n NAME, --name NAME  file name
  -d DST, --dst DST     destination file path
  -pr RDTPROTOCOL, --RDTprotocol RDTPROTOCOL
                        stop_and_wait (sw) or selective_repeat (sr)
  -f, --files           show available files to download
```

3.2. Handshake

La comunicación comienza con un "three-way handshake". Este es un proceso fundamental para asegurar una conexión confiable entre el cliente y el servidor y garantiza que ambas partes estén listas para intercambiar datos antes de comenzar la transmisión de información.

Este intercambio de mensajes inicial consiste en tres pasos fundamentales:

1. **Solicitud de conexión (HI)**: El cliente envía un mensaje al servidor con el flag HI. Esto indica que el cliente desea establecer una conexión y está solicitando que el servidor también lo haga.
2. **Aceptación de conexión (HI-ACK)**: Si el servidor está dispuesto a aceptar la conexión, responde al cliente con un mensaje en el que establece el flag HI-ACK. Este indica que el servidor está listo para recibir datos y que ha recibido correctamente la solicitud de conexión del cliente.
3. **Confirmación de conexión (HI-ACK)**: Finalmente, el cliente responde al servidor con un mensaje en el que establece el flag HI-ACK. Con este mensaje se confirma la aceptación de la conexión por parte del servidor y que está listo para comenzar la transmisión de datos. A partir de este punto, la conexión se considera establecida y ambas partes pueden comenzar a intercambiar datos de manera confiable.

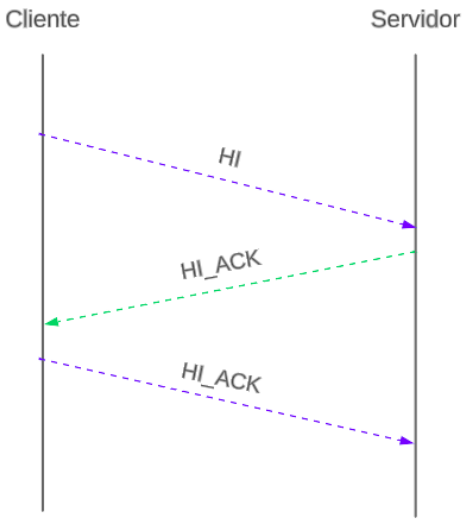

Figura 1: Three-way-handshake

3.2.1. Comportamiento en caso de packet loss

Tenemos tres escenarios:

1. Se pierde el mensaje con el flag HI: En este caso, luego de enviarlo, el cliente tiene un timeout establecido. Cuando se llega a ese tiempo, se reintenta con hasta un máximo de 10 veces.
2. Se pierde el mensaje con el flag HI_ACK del servidor: Este escenario queda cubierto con el mismo mecanismo del item 1. Si el servidor envía HI_ACK pero nunca le llega al cliente, saltará el timeout del primer caso y se volverá a enviar HI.

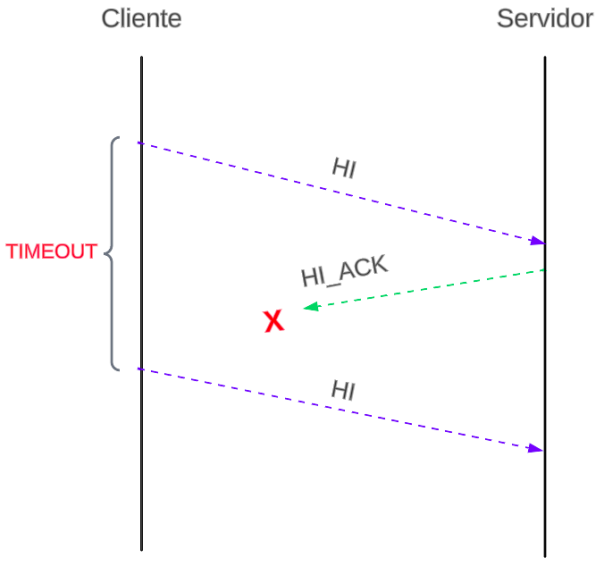

Figura 2: Se pierde HI-ACK del servidor

3. Se pierde el mensaje con el flag HI_ACK del cliente: Si se perdiera este mensaje, el cliente continúa con sus operaciones. Sin embargo, el servidor no responde hasta no recibir el correspondiente HI_ACK. Tomamos la decisión entonces de tener un timeout en el receive del cliente, que llegado el caso, corta su accionar y finaliza la conexión. Es decir, no se vuelve a enviar el HI_ACK al servidor, sino que si este paquete se pierde, finaliza el programa.

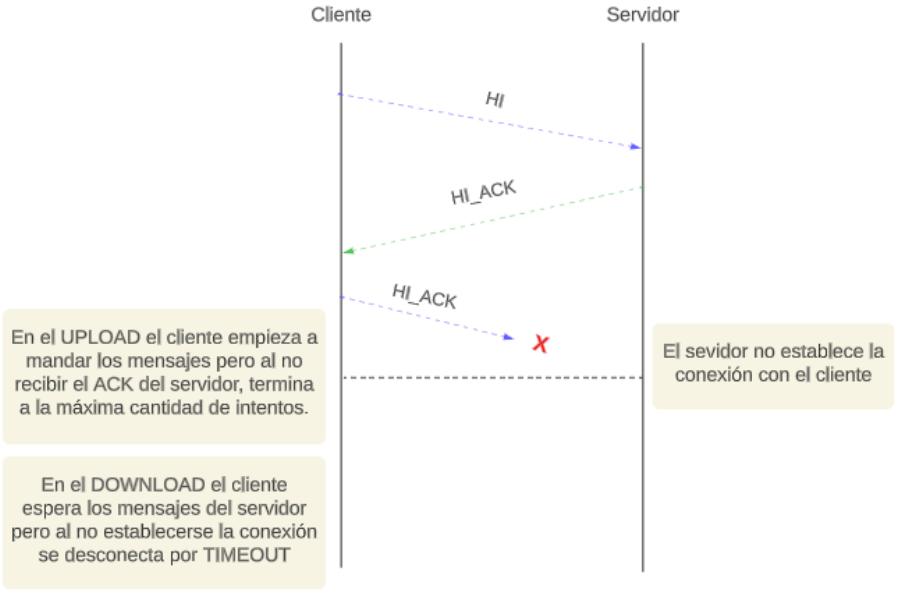

Figura 3: Se pierde HI-ACK del cliente

3.3. Stop and Wait

La aplicación desarrollada permite hacer la subida o la descarga de archivos con distintos protocolos que aseguran que la transmisión sea confiable. Uno de ellos es el protocolo de Stop and Wait.

Este protocolo consiste en ir enviando de a un mensaje y no enviar el siguiente hasta no haber recibido un ACK correspondiente del último enviado.

En la implementación realizada en este trabajo, el ACK que se envía es del siguiente mensaje que espera recibir, es decir, si se envía un mensaje ACK con un ack_number 3, significa que recibió correctamente los paquetes 0, 1 y 2, y el siguiente que espera recibir corresponde con el paquete 3.

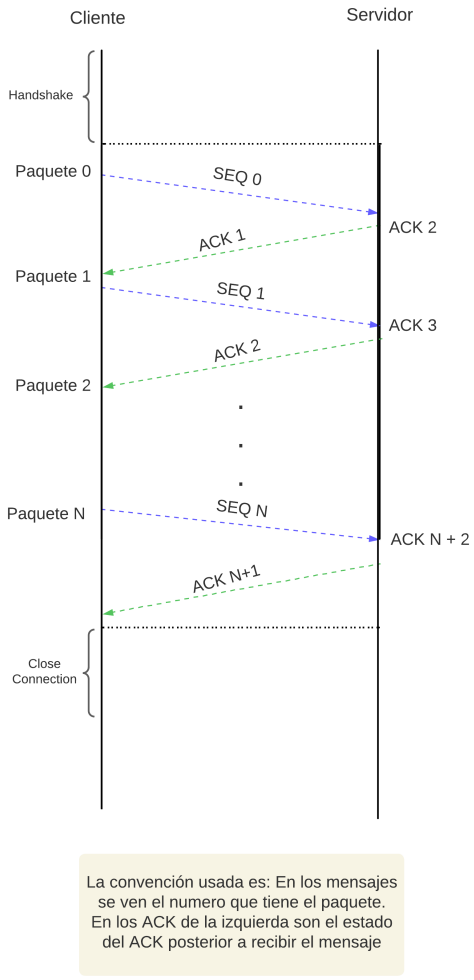

Figura 4: Stop and Wait

Hay algunas situaciones que tuvieron que tener en cuenta a la hora de desarrollar este protocolo:

1) De un lado de la comunicación se envía un mensaje UPLOAD o DOWNLOAD, pero no llega al otro lado. En este caso, el que está esperando recibir espera indefinidamente y finalmente agota su tiempo de espera en el socket. Entonces, el que mandó el primer mensaje reenvía el mensaje con el comando UPLOAD o DOWNLOAD con el mismo número de secuencia, y en este punto, el receptor en este caso no ha procesado nada.

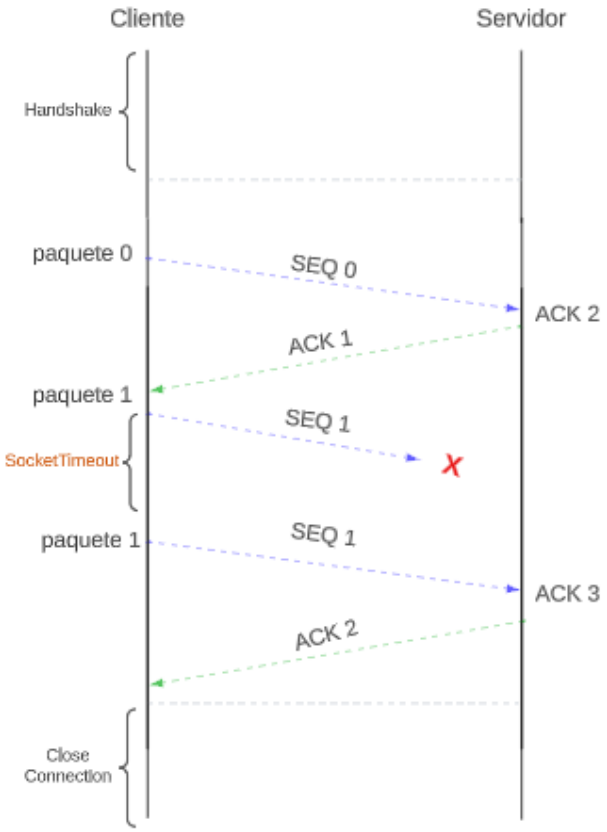

Figura 5: Se muestra qué pasa cuando se pierde un mensaje en Stop and Wait

2) De un lado de la comunicación se manda un mensaje con el comando UPLOAD o DOWNLOAD. Del otro lado se responde con un ACK, pero la respuesta ACK se pierde en la red. En esta situación, el que recibió y respondió al mensaje aumenta su número de ACK, pero dado que el ACK original nunca llega al otro lado, el número de secuencia del que mandó el mensaje se mantiene sin cambios. Por lo tanto, cuando el emisor del primero vuelva a mandar el mensaje, el receptor recibe un número de secuencia que es dos unidades menor que su ACK. En este caso, el receptor responde al emisor devolviendo el número de secuencia + 1 y no realiza ninguna otra acción porque ya tiene ese paquete. Si llega un número de secuencia que es solo 1 unidad mayor que el ACK, el receptor original ejecuta su operación normalmente. Es importante destacar que, debido al escenario 1), nunca llegará un número de secuencia mayor que el ACK.

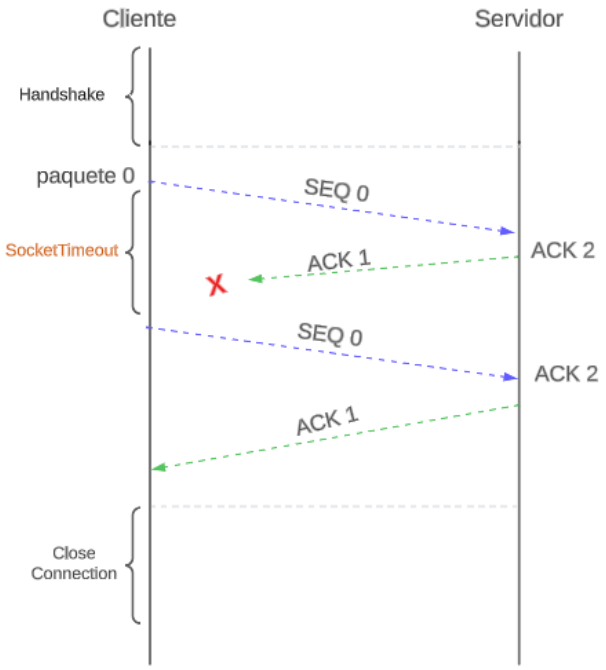

Figura 6: Se muestra qué pasa cuando se pierde un ACK de un mensaje en Stop and Wait

3.4. Selective repeat

El otro protocolo implementado para hacer un envío confiable de la información es el llamado Selective Repeat. Para correr el programa con este procolo se debe especificar en los argumentos con -pr sr.

En este se utilizan dos ventanas: una en el lado del emisor y otra en el lado del receptor. Estas ventanas tienen funciones específicas en el proceso de comunicación.

La ventana del emisor cumple dos roles principales. Primero, se utiliza para rastrear las confirmaciones de recepción (ACKs) de los paquetes enviados. Segundo, limita la cantidad de paquetes "en vuelo "al restringir cuántos paquetes pueden ser enviados antes de recibir ACKs correspondientes.

Cuando el emisor envía mensajes, estos deben estar dentro de su ventana. A medida que recibe los ACKs de los mensajes enviados, los marca como recibidos, lo que le permite enviar más mensajes. En particular, si el ACK que llega es del paquete correspondiente al send_base (es decir, el primero de la ventana), se mueve. Este mecanismo garantiza que no se sature la red con una cantidad excesiva de paquetes no confirmados.

Por otro lado, el receptor también utiliza una ventana que cumple una función esencial. Esta ventana se encarga de llevar un registro de los paquetes esperados. Si un mensaje llega fuera del rango de su ventana, el receptor lo descarta. En cambio, si el mensaje está dentro de la ventana, el receptor siempre envía un ACK al emisor y almacena la información en un buffer en caso de no ser el paquete en el orden esperado.

A medida que llegan los mensajes que el receptor espera, amplía su ventana para acomodarlos. Cuando finalmente recibe un mensaje en el orden esperado, procesa el buffer para asegurar que los datos se procesen en el orden correcto, garantizando así la integridad y el orden de la comunicación.

Este enfoque de ventanas y seguimiento de ACKs en el protocolo Selective Repeat es esencial para lograr una transmisión confiable y mucho más eficiente de datos en redes con posibles pérdidas de paquetes a comparación de Stop and Wait.

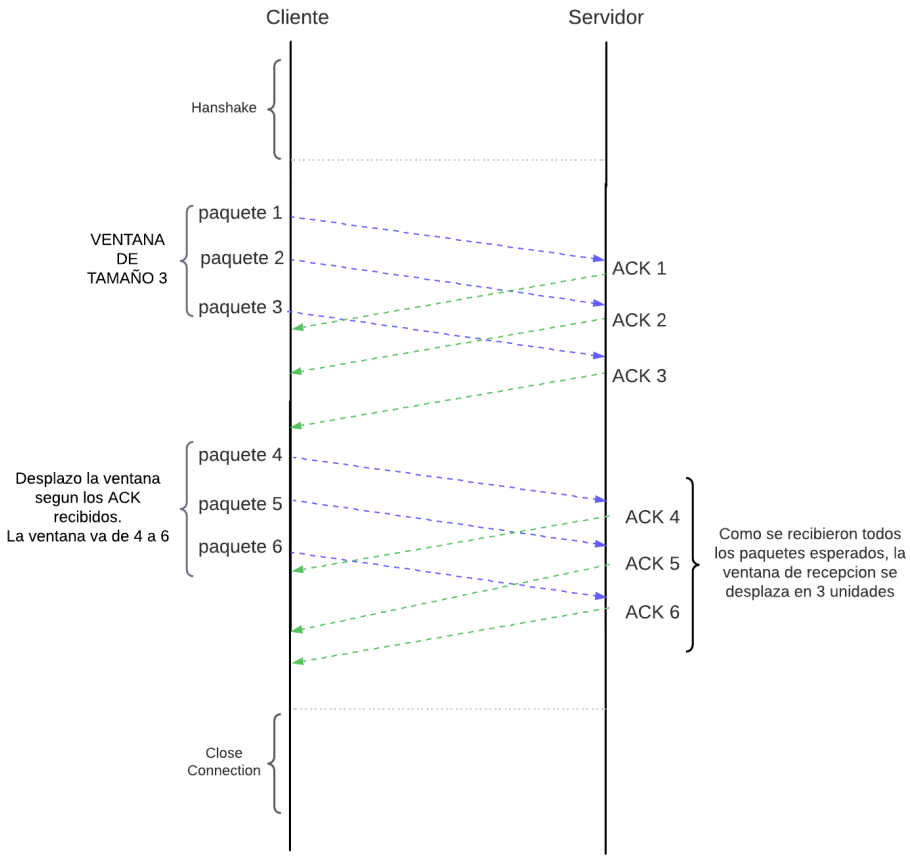

Figura 7: Selective Repeat

Cuando se pierde un paquete, detectado gracias a su propio timeout, se envía solamente el que se perdió.

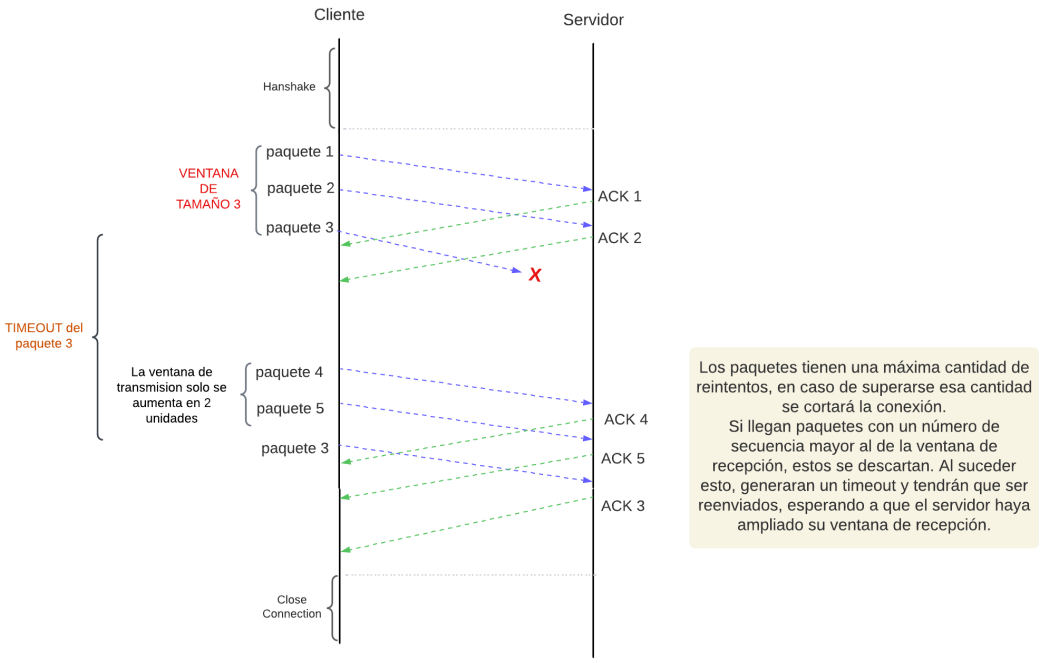

Figura 8: Selective Repeat cuando se pierde un mensaje

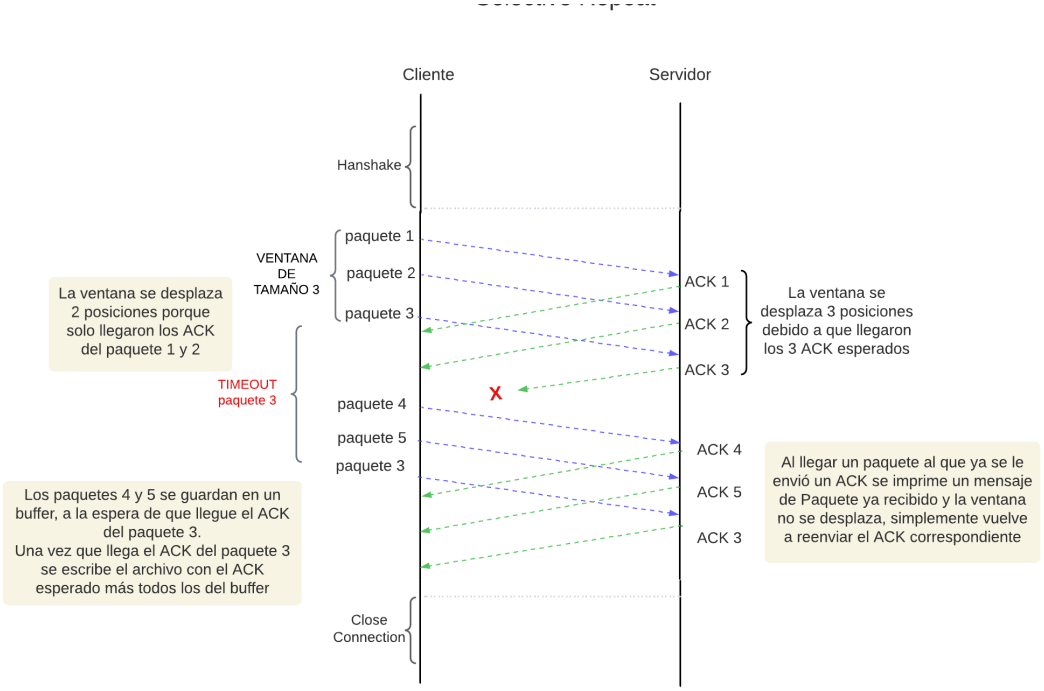

Figura 9: Selective Repeat cuando se pierde el ACK de un mensaje

3.5. Cierre de conexión

Para finalizar la conexión utilizamos un mensaje con el flag CLOSE y otro con CLOSE_ACK en respuesta. Tanto en Selective Repeat como en Stop and Wait el procedimiento es el mismo. El encargado de enviar el cierre es siempre el que envía el archivo. En DOWNLOAD es el servidor y en UPLOAD el cliente. Este mensaje final se envía cuando todos los paquetes han sido enviados y reconocidos por la otra parte.

¿Qué sucede si se pierde el CLOSE? La parte que envía este mensaje tiene un timeout para recibir el CLOSE_ACK con un máximo de reintentos. Si el CLOSE no llegó, tampoco lo hará el CLOSE_ACK, por lo tanto se llegará al tiempo establecido y se enviará de nuevo el CLOSE.

3.6. Diagrama de threads y procesos

La comunicación a nivel aplicación se puede ver representada en la siguiente imagen.

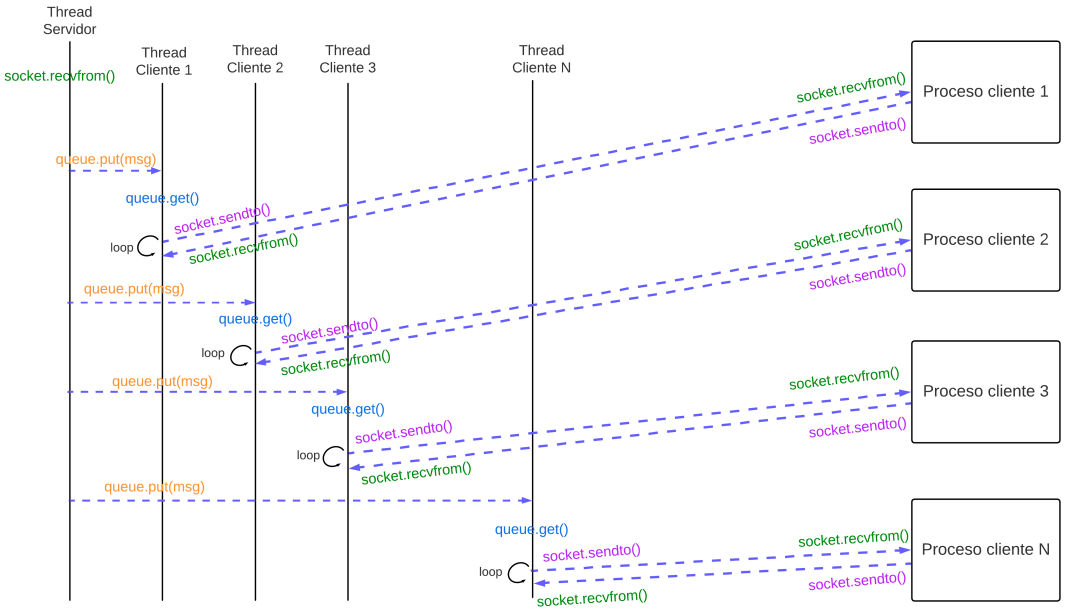

Figura 10: Diagrama de threads y procesos

4. Pruebas

Para testear nuestro trabajo contamos con 2 métodos complementarios:

- Ejecución + hash: Para ejecutar ya sea UPLOAD o DOWNLOAD debemos levantar el servidor en una terminal y en otra algún cliente:

```
# start server en una terminal
python3 start-server.py -v

# upload selective repeat en otra terminal
python3 upload.py -v -s <path_archivo_a_subir> -pr sr

# upload stop and wait en otra terminal
python3 upload.py -v -s <path_archivo_a_subir>

# download selective repeat en otra terminal
python3 download.py -v -d <path_guardar> -pr sr -n <archivo_en_server_storage>

# download stop and wait en otra terminal
python3 download.py -v -d <path_guardar> -n <archivo_en_server_storage>
```

  Para verificar que el archivo descargado o subido es igual al original, podemos calcularle la función de hash md5sum y ambos resultados deberían ser idénticos:

```
    $ md5sum <path_archivo_original> <path_nuevo_archivo>
```

- Ejecución + Wireshark: Para constatar que los mensajes de nuestro protocolo son efectivamente como detallamos en este informe, necesitamos capturar paquetes con la herramienta Wireshark. Para esto, hicimos un plugin que parsea los bytes de los mensajes de la capa de

aplicación a los campos de nuestro protocolo. El mismo se encuentra en el repositorio con el nombre *dissector.lua*. Para utilizarlo, hay que seguir estos pasos:

1. Tener instalado Wireshark
2. Ir a Ayuda > Acerca de Wireshark > Carpetas y hacer click en la Ubicación del item Complementos personales de Lua. Darle aceptar. En caso de no tener la carpeta creada, la creará.
3. Copiar y pegar el disector en esa carpeta y ya lo utilizará en las próximas capturas.

Una vez configurado el plugin, podemos ejecutar nuestro programa ya sea con UPLOAD o DOWNLOAD teniendo Wireshark de fondo capturando nuestra interfaz.

Por otro lado, para probar con pérdida de paquetes, utilizamos Comcast con esta configuración:

```
$ go run comcast.go --device=lo --packet-loss=10%
```

El device puede depender de cada uno, podemos ver cuál utilizar ejecutando:

```
$ ip a
```

y seleccionando el que aparece en uso.

4.1. Mediciones

A continuación se adjunta un análisis comparativo de la transferencia de datos usando Stop And Wait o Selective Repeat con y sin packet loss, medida en segundos. Se testó usando únicamente el UPLOAD.

Stop and Wait

	3.2MB	5.2MB
0 % Package loss	1.982s	3.137s
5 % Package loss	5,485s	8,764s

Selective Repeat

	3.2MB	5.2MB
0 % Package loss	3,979s	6,785s
5 % Package loss	6,843	9,367

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

La arquitectura Cliente-Servidor es un modelo en el que un host, conocido como "Servidor", está siempre activo y listo para atender solicitudes de otros hosts llamados " Clientes". En este modelo, los Clientes no se comunican directamente entre sí, sino que interactúan exclusivamente con el Servidor.

El Servidor actúa de manera pasiva, esperando solicitudes de los Clientes para proporcionarles recursos o servicios. Esto se asemeja a un servicio de "tirar" o "pull", donde los Clientes solicitan información o ejecutan acciones, y el Servidor responde en función de esas solicitudes.

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

La capa de aplicación es aquella que permite la comunicación entre los procesos de las aplicaciones a través de la red. Esta capa proporciona un conjunto de servicios y protocolos que permiten a las aplicaciones en los diferentes end systems intercambiar datos de manera efectiva y fiable. Por ejemplo, definiendo cómo será el formato de los mensajes: podrían ser variables o fijos, tener diferentes campos en el header, etc. Los protocolos de capa de aplicación son utilizados tanto por los dispositivos de origen como de destino durante una sesión de comunicación, por lo tanto para que las comunicaciones se lleven a cabo correctamente, deben ser compatibles.

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

El protocolo desarrollado en este trabajo está explicado en detalle en la sección 3 de este informe.

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

En cuanto a UDP (User Datagram Interface), este procolo ofrece pocos servicios aunque suficientes para lograr una comunicación básica:

- Multiplexado y demultiplexado, permitiendo de esta forma la comunicación proceso a proceso.
- Control de integridad de los mensajes a través del checksum.

TCP (Transfer Control Protocol), por otro lado, además de ofrecer los servicios que ofrece UDP, también cuenta con:

- RDT (Reliable Data Transfer), que implica que los mensajes que llegan al host receptor se reciben en orden, con toda la información íntegra, sin errores ni duplicados. Además ofrece control de flujo, es decir que ajusta la velocidad a la que manda los mensajes para no sobrecargar al emisor, evitando la pérdida de paquetes por overflow.
- Control de congestión, usa algoritmos y mecanismos para evitar y controlar la congestión de la red, ajustando dinámicamente la velocidad de transmisión de datos en función de las condiciones de la red. Esto evita que haya pérdida de datos y problemas de rendimiento.

6. Dificultades encontradas

Enumeramos algunas dificultades con las que nos encontramos a lo largo del desarrollo del trabajo:

- Definición los campos de los mensajes necesarios para realizar una buena comunicación entre cliente y servidor.
- Definición los valores de los timeout, dado que dependiendo de los tamaños de archivo podían quedar cortos.
- Manejo de la pérdida de paquetes para el correcto envío del archivo.
- Manejo de los timeouts individuales de cada mensaje en Selective Repeat. Esto significó el uso de threads y manejo de race conditions.
- Determinación del tamaño de ventana en Selective Repeat. Utilizamos un tamaño fijo para la ventana de recepción para evitar complicaciones dado que sino, tendríamos que solicitarle al cliente que nos indique cual será el tamaño del archivo que nos quiere enviar.
- El lenguaje (Python) no era muy amigable para encontrar errores, sobre todo por ser de tipado dinámico.

7. Conclusión

En este trabajo práctico, desarrollamos un protocolo de aplicación para la transferencia de archivos utilizando el protocolo de transporte UDP con extensiones para lograr una comunicación confiable. Implementado dos versiones de este protocolo, una basada en el esquema Stop and Wait y otra en el esquema Selective Repeat. A lo largo de este proyecto, abordamos varios desafíos relacionados con la definición de los mensajes, el manejo de timeouts, la pérdida de paquetes y otros aspectos de la comunicación cliente-servidor.

Logramos crear un programa que permite a los usuarios cargar y descargar archivos de manera confiable a través de una conexión UDP. Además, hemos realizado pruebas suficientes, incluyendo escenarios de pérdida de paquetes con archivos de diferentes tamaños, para validar la confiabilidad y robustez de nuestro protocolo.

En resumen, este proyecto nos ha permitido comprender en profundidad los desafíos asociados con la implementación de un protocolo de aplicación y nos brindó una buena experiencia en el diseño y desarrollo de sistemas de comunicación confiables.