Redes (TA048/75.43/75.33/95.60)
1C - 2025

Trabajo Práctico 1 - File Transfer

Grupo 20

Fecha de entrega: 08/05/2025

Integrantes:
Santiago Marczewski - 106404

Índice

1. Introducción

En este informe se detalla una aplicación de red para la transferencia de archivos con una arquitectura cliente-servidor. La misma utiliza un protocolo RDT (del inglés Reliable Data Transfer, RDT) para la transmisión de los datos de manera confiable, el cual está implementado sobre el protocolo de capa de transporte UDP para sortear las deficiencias del mismo para lograr transmitir de manera confiable los archivos sin importar las condiciones de la red.

Además de cuestiones más técnicas como pueden ser las especificaciones del protocolo implementado o pruebas de performance, se incluirán también algunas cuestiones más teóricas del tema en general, así como también algunos detalles sobre la experiencia desarrollando la aplicación.

2. Hipótesis y suposiciones realizadas

En un principio se hicieron las siguientes suposiciones:

1. Los segmentos UDP pueden **no llegar** a destino.
2. Los segmentos UDP pueden llegar a destino **en cualquier orden.**
3. Los segmentos UDP que lleguen a destino **no estarán corruptos** gracias al checksum.

Además como hipótesis podemos mencionar que esperamos que la aplicación responderá mejor a condiciones desfavorables en la red si utiliza el mecanismo de error-recovery de *selective repeat,* frente al uso de *stop & wait*.

3. Implementación

3.1 Estructura de los paquetes

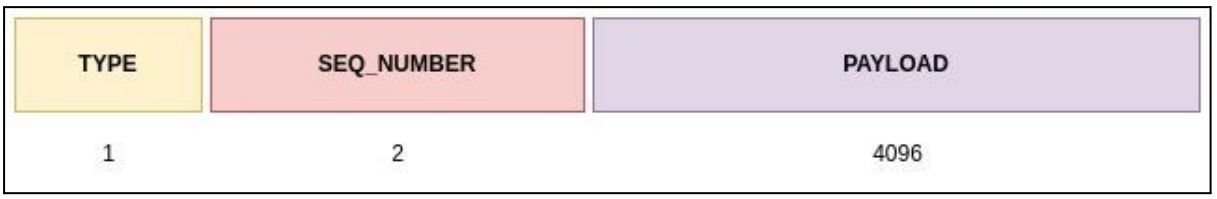

Diagrama 3.1

Tenemos diversos tipos de paquetes:

- **DOWNLOAD**: Este paquete inicia la conexión desde el host cliente para iniciar una descarga, contiene como payload el nombre del archivo a descargar, lo que nos permite chequear que sea un archivo válido y devolver el mensaje de ERROR o DATA (implícitamente diciendo que todo está en orden) según corresponda.

- **UPLOAD**: Cumple el mismo propósito que DOWNLOAD pero para iniciar una subida al servidor, el servidor le devuelve ACK (dando pie a que podamos enviar DATA) o ERROR.

- **ACK**: Sirve para notificar del recibimiento de un paquete, su número de secuencia en tándem con el de los paquetes DATA nos permite mantener un orden de paquetes e identificar faltantes.

- **DATA**: Contiene en su payload los datos tanto para enviar al servidor en caso de hacer una subida, como para recibirlos de él en caso de una descarga.

- **ERROR**: Contiene un mensaje de error y notifica al host cliente que hubo algún error y el mensaje de error correspondiente está contenido en su payload.

- **CLOSE**: Sirve para notificar que el host en rol de sender terminó de enviar el archivo, por lo que saber que lo recibió efectivamente el host en rol de receiver nos asegura que recibió la totalidad del archivo.

3.2 Establecimiento de la conexión:

A continuación se detalla el proceso de establecimiento de la conexión (o *handshake*) tanto para la subida como para la descarga.

Conexión inicial para subida y descarga

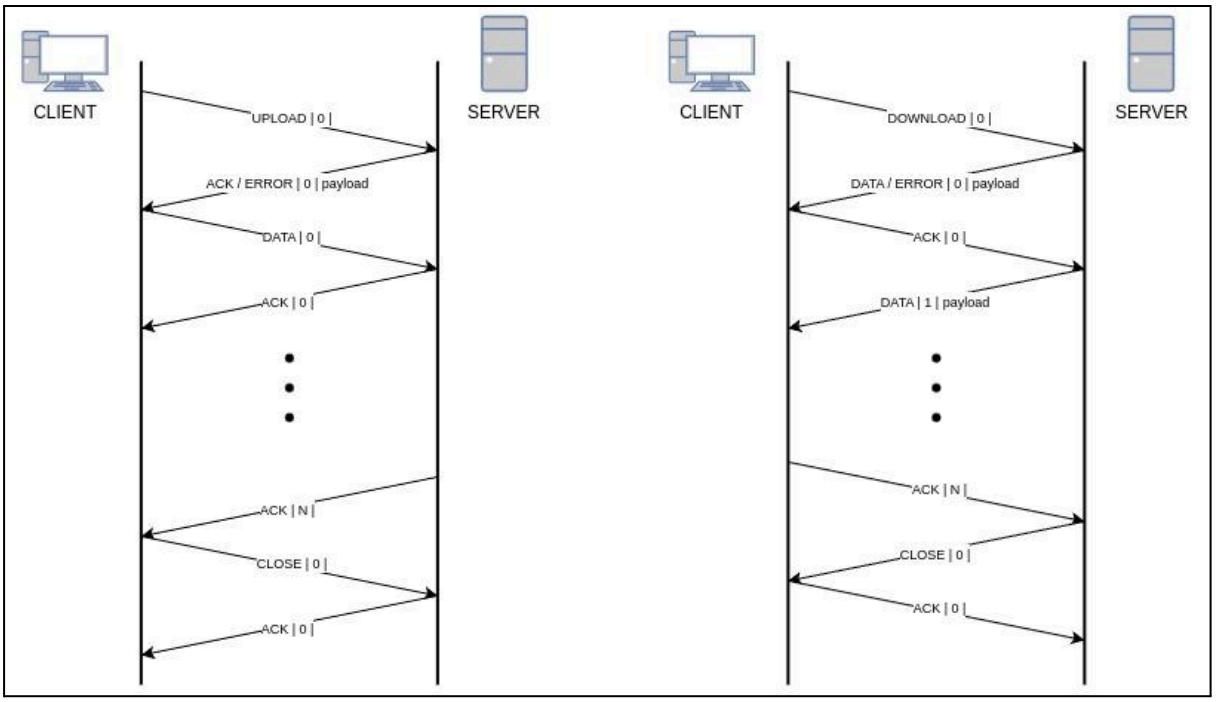

Diagrama 3.2

En el caso de la subida el host cliente (en este caso en rol de sender) envía primero un paquete UPLOAD hasta recibir un ACK o un ERROR. En caso de recibir un ACK podemos empezar a enviar DATA al servidor (en rol de receiver) sabiendo que la va a estar esperando. Por otro lado si nos devuelven un ERROR será porque estamos intentando subir un archivo con nombre idéntico a uno que ya se encuentra en el almacenamiento del servidor.

En el caso de la descarga o bajada, el cliente (en este caso en rol de receiver) envía primero un paquete DOWNLOAD hasta recibir un paquete DATA o un ERROR. En caso de recibir un paquete DATA asumimos que está todo bien y nos ponemos a recibir los subsiguientes paquetes DATA del archivo que nos enviará el servidor. En caso de recibir un ERROR esto significa que intentamos descargar un archivo cuyo nombre no coincide con ninguno de los presentes en el almacenamiento del servidor.

3.3 Comportamiento general y recuperación en Stop & Wait:

Se puede dar durante la ejecución que se pierda durante la ejecución un paquete. En cualquier caso de pérdida de paquetes, el manejo se hace a través de los sockets UDP, al ser bloqueante la lectura y tener *timeouts* configurables, nos dan la posibilidad de saber cuando algo se perdió en el camino.

En caso de haber enviado un ACK y estar esperando el siguiente DATA, si el socket donde lo espero hizo *timeout* sabemos que se perdió y debemos reenviar el ACK. El caso inverso es análogo, si hace timeout el socket dónde estoy esperando un ACK sé que debo reenviar el paquete DATA.

El siguiente diagrama es un ejemplo de recuperación después de la pérdida de un ACK en el caso de una operación de descarga:

Descarga en Stop & Wait:

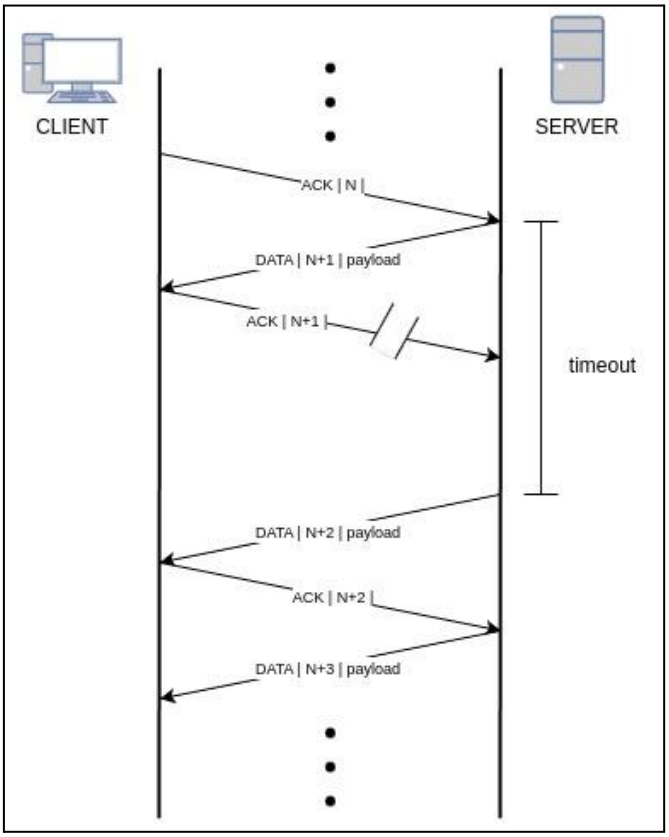

Diagrama 3.3

3.4 Comportamiento general y recuperación en Selective Repeat:

El siguiente diagrama detalla el comportamiento general y la recuperación ante una pérdida de paquete (en este caso uno DATA) para el protocolo de *selective repeat*:

Descarga en Selective Repeat:

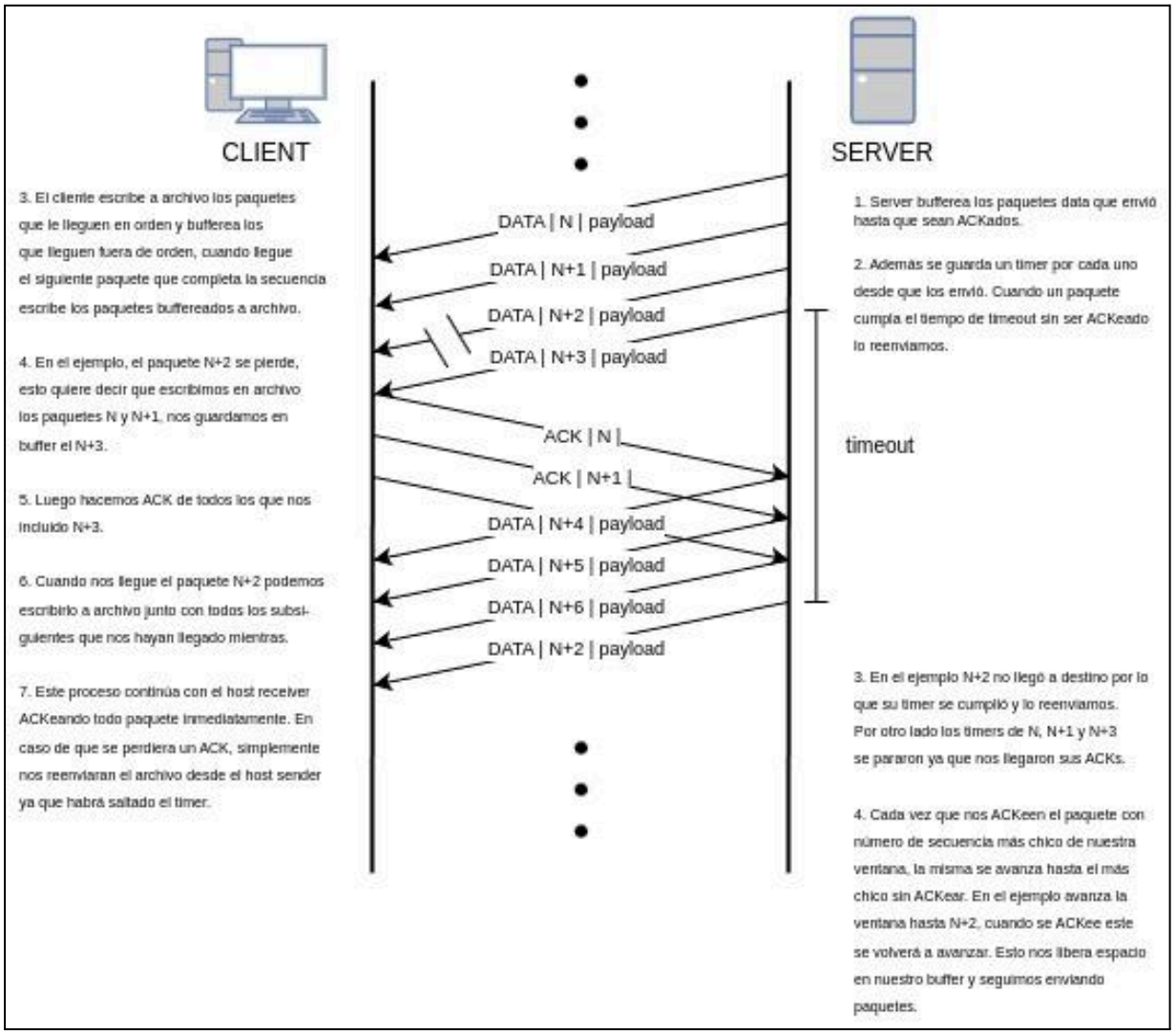

Diagrama 3.4

Es un protocolo bastante simple, en el lado del host receiver simplemente esperamos los paquetes DATA, en caso de llegar en orden los vamos escribiendo a archivo. En caso de que se pierda un paquete en el camino, vamos a guardarnos todos los subsiguientes dentro del tamaño de la ventana (tamaño también de nuestro buffer) en un buffer de paquetes. Una vez que en el lado del host sender haya un timeout del paquete que nos falta, nos lo van a reenviar. Cuando tengamos nuestro paquete faltante podemos escribir completos y en orden toda la secuencia que tengamos en el buffer hasta el siguiente paquete que nos falta.

Para el caso del host sender es un poco más complejo, tendremos un buffer de los paquetes DATA en vuelo (los que enviamos y aún no han sido ACKeados) y un array de timers para cada uno de ellos.

Cuando haya un timeout para uno de los paquetes aun en vuelo asumimos que no llegó a destino y lo reenviamos, además reiniciamos su timer. En caso de que nos ACKeen un paquete y este sea el menor de nuestro buffer podemos “adelantar” la ventana. Esto significa que sabemos que esos paquetes llegaron y podemos hacer espacio para paquetes DATA nuevos a mandar.

4. Pruebas

Se hicieron pruebas de performance a la aplicación con distintas configuraciones para comparar los dos protocolos en distintas condiciones de red (simuladas a través de mininet).

Stop & Wait - Download

Tamaño de archivo (MB)	% de pérdida	Tiempo (segundos)
5,4	0	0,09
5,4	5	9,27
5,4	10	20,13
5,4	25	88,87

Tabla 4.1

Stop & Wait - Upload

Tamaño de archivo (MB)	% de pérdida	Tiempo (segundos)
5,4	0	0,08
5,4	5	9,48
5,4	10	20,18
5,4	25	86,20

Tabla 4.2

Selective Repeat - Download

Tamaño (MB)	% de pérdida	Tiempo (segundos)
5,4	0	1,24
5,4	5	3,11
5,4	10	6,13
5,4	25	64,51

Tabla 4.3

Selective Repeat - Upload

Tamaño (MB)	% de pérdida	Tiempo (segundos)
5,4	0	1,99
5,4	5	3,66
5,4	10	6,13
5,4	25	64,38

Tabla 4.4

4.1 Análisis de las pruebas

Podemos observar en los resultados de las pruebas que el protocolo *stop & wait* es más rápido en condiciones de red perfectas que *selective repeat*. Pero se ve que fuera de la situación ideal *selective repeat* es el protocolo que responde mejor.

Armando una tabla comparativa del porcentaje de mejora de un protocolo frente al otro podemos observar esto con más claridad.

Comparación Stop & Wait vs. Selective Repeat para archivo de 5,4 MB

Stop & Wait (% de mejora)	**Selective Repeat** (% de mejora)
+ 1900%	- 1900%
- 277%	+ 277%
- 329%	+ 329%
- 136%	+ 136%

Tabla 4.5

Se ve una tendencia de *selective repeat* a ir perdiendo su efectividad frente a *stop & wait* a medida que aumenta la pérdida de paquetes. Aunque cabe destacar que de cualquier manera sigue teniendo una ventaja significativa cuando la red presenta pérdida de paquetes.

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

En la arquitectura cliente-servidor tenemos un host *servidor* que se mantiene constantemente activo, atendiendo las solicitudes de otros hosts *clientes*. Incluso mediando entre clientes que quieren comunicarse entre sí, es decir que los hosts clientes nunca se comunican directamente entre ellos. Esto

implica que el host servidor debe mantener una dirección IP fija y conocida, para que los clientes sepan dónde encontrarlo y poder enviarle paquetes.

Un ejemplo de esta arquitectura puede ser un host servidor web que atiende las solicitudes de los distintos hosts clientes, por ejemplo navegadores que solicitan objetos para mostrar una página web.

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

La función de un protocolo de capa de aplicación es definir cómo se comunican entre sí los procesos de una aplicación, que se están ejecutando en hosts diferentes, haciéndolo esencial para las aplicaciones de red. Entre las cosas más importantes que define se encuentran los tipos de mensajes, los campos que contienen, su sintaxis y las reglas para el flujo de los mismos.

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

El protocolo de aplicación desarrollado se encuentra detallado en la **sección 3** del 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?¿Cuándo es apropiado utilizar cada uno?

UDP es un protocolo muy simple y ofrece solamente un servicio *best-effort*. Esto significa que no da garantías de que los segmentos lleguen efectivamente a destino ni que lo hagan en el orden enviado, aunque sí podemos decir que garantiza que los segmentos no lleguen corruptos. No mantiene un estado compartido entre los hosts.

Por otro lado TCP ofrece más servicios. Primeramente mantiene un estado compartido entre los hosts al realizar un *handshake* inicial, y además garantiza que los segmentos lleguen a destino, que lo hagan en el orden enviado y que no lleguen corruptos. Además TCP ofrece un servicio de control de congestión que reacciona acorde al estado de la red para mitigar sus impactos negativos.

UDP es más apropiado si se busca flexibilidad, ya que tenemos más control sobre los segmentos. Además su simplicidad resulta atractiva para aplicaciones que busquen más responsividad, nos permite saltarnos el control de congestión de TCP, no hace falta mantener una conexión y sus segmentos son más ligeros. Un ejemplo pueden ser las aplicaciones de streaming de video.

TCP es una mejor opción para aplicaciones que requieran transferencia de datos confiables sin importar las condiciones de la red. Un ejemplo pueden ser páginas web.

6. Dificultades encontradas

A nivel implementación una dificultad fue encontrar valores buenos para los timeouts de los diferentes sockets utilizados, siempre buscamos valores lo más bajos posibles para no perder tiempo en innecesariamente en la espera pero si son demasiado bajos llega un punto en el cual el trade-off se vuelve malo ya que empezamos a dar por pérdidos paquetes que no lo están, lo cual baja la

performance. Los valores utilizados se seleccionaron en base a prueba y error para encontrar un punto medio.

También podemos mencionar que no se logró una verdadera compatibilidad entre los dos protocolos, es decir, si uno de los hosts está en modo *stop & wait* y el otro en *selective repeat*, no se puede garantizar que la transferencia de archivos sea confiable.

7. Conclusión

Se pudo observar que nuestra hipótesis inicial era correcta, efectivamente el protocolo selective repeat responde mucho mejor a condiciones desfavorables de red que stop & wait. También podemos decir que se logró con éxito la transmisión confiable de datos, y por ende de archivos, con ambos protocolos.