Universidad de Buenos Aires
Facultad de Ingeniería

Redes - Hamelin

Trabajo Práctico N.º 1: “File Transfer”

2do Cuatrimestre 2025

Alumno:

- Ezequiel Martín Aragón 110643
- Martín Castro 109807
- Nicolás Khalil Chaia 110768
- Gonzalo Nicolas Crudo 110816
- Franco Guardia 109374

Introducción

El presente trabajo tiene como finalidad el diseño e implementación de una aplicación de red bajo una arquitectura cliente-servidor, con el objetivo de comprender en profundidad el funcionamiento de la comunicación entre procesos a través de la red y los servicios que la capa de transporte brinda a la capa de aplicación.

Para alcanzar este objetivo, se emplea la interfaz de **sockets** en el lenguaje **Python**, utilizando **UDP** como protocolo de transporte. Dado que este protocolo no garantiza la entrega confiable de datos, será necesario implementar un protocolo de aplicación que contemple mecanismos de **Reliable Data Transfer (RDT)**. En particular, se abordarán dos estrategias: **Stop & Wait** y **Go-Back-N**, las cuales permitirán asegurar la correcta transferencia de información aún en condiciones de pérdida de paquetes.

La aplicación desarrollada permitirá la transferencia de archivos binarios entre cliente y servidor mediante las operaciones de **UPLOAD** y **DOWNLOAD**. Asimismo, se diseñará un protocolo de aplicación que defina los mensajes intercambiados y gestione posibles condiciones de error. Para validar su funcionamiento, se realizarán pruebas bajo distintos escenarios de red simulados con **Mininet**, incluyendo situaciones con pérdida de paquetes y múltiples clientes concurrentes.

Finalmente, el trabajo incluirá un análisis comparativo del desempeño de las versiones implementadas de Stop & Wait y Go-Back-N bajo diferentes condiciones, con el fin de evaluar las ventajas y limitaciones de cada mecanismo.

Hipótesis y suposiciones realizadas

Hipótesis

- El archivo a subir al servidor no puede superar los 5MB (5.242.880 bytes)
- Si el cliente quiere subir un archivo que ya se encuentra en el servidor entonces lo sobreescribe (y si el cliente quiere descargar un archivo que ya tiene lo sobreescribe también en su propio lugar de almacenamiento)
- No hay límite en la cantidad de conexiones simultáneas al servidor

Suposiciones

- Tanto el servidor como el cliente tienen una carpeta local asignada para cada uno donde estarán los archivos que correspondan a su almacenamiento (del lado del servidor aquellos que el cliente haya subido y del lado del cliente aquellos que haya descargado o que ya tenía inicialmente)
- Tanto del lado del cliente como del servidor se hacen 5 reintentos de enviar un paquete cuando el otro lado no responde, luego de eso se considera que la conexión se cayó y no se vuelve a intentar

- La ventana deslizante en Go-Back-N tiene un total de 5 paquetes en vuelo que puede estar esperando a la vez
- El tamaño de los paquetes es aproximadamente fijo y se calcula como la suma de un valor de MSS prefijado y el tamaño del header

Implementación

El protocolo implementado consta de:

1. Header fijo de 12 bytes que contiene:
    a. *next_seq*: Número de secuencia siguiente.
    b. *is_last*: Indica si le siguen más paquetes, si no hay se setea en 1, caso contrario, 0.
2. Campo Data fijo de 1388 bytes, que surge de restar el MSS con el tamaño del Header.

Handshake

El Handshake representa el inicio de la comunicación entre los dos hosts involucrados en la transferencia de un archivo.

En este proceso, el cliente envía un mensaje de tipo INIT, completando el tamaño correspondiente al Header. En la sección de Data, se incluyen los siguientes datos:

- El mecanismo de confiabilidad seleccionado (Stop & Wait o Go-Back-N),
- La acción a realizar (UPLOAD o DOWNLOAD),
- Y el nombre del archivo asociado.

El cliente realiza hasta cinco intentos de envío del mensaje INIT (MAX_WINDOWS_RETRIES y MAX_WAIT_PACKETS). Si no recibe respuesta en ninguno de ellos, la aplicación finaliza la ejecución.

Por su parte, el servidor implementa el siguiente mecanismo:

1. Detecta la conexión de un nuevo cliente.
2. Crea una cola (Queue) dedicada para la comunicación con dicho cliente.
3. Lanza un hilo (Thread) exclusivo para manejar la interacción.
4. Parsea el mensaje recibido, por ejemplo: ["SW", "UPLOAD", "documento.bin"].
5. Instancia el protocolo correspondiente (Stop & Wait o Go-Back-N).

6. Envía al cliente un mensaje de confirmación ACK0, indicando la correcta inicialización de la sesión.

Stop & Wait:

El protocolo Stop & Wait se basa en un mecanismo sencillo de control de flujo y confiabilidad. En cada iteración, el emisor envía un solo paquete y espera la confirmación (ACK) correspondiente antes de proceder con el siguiente.

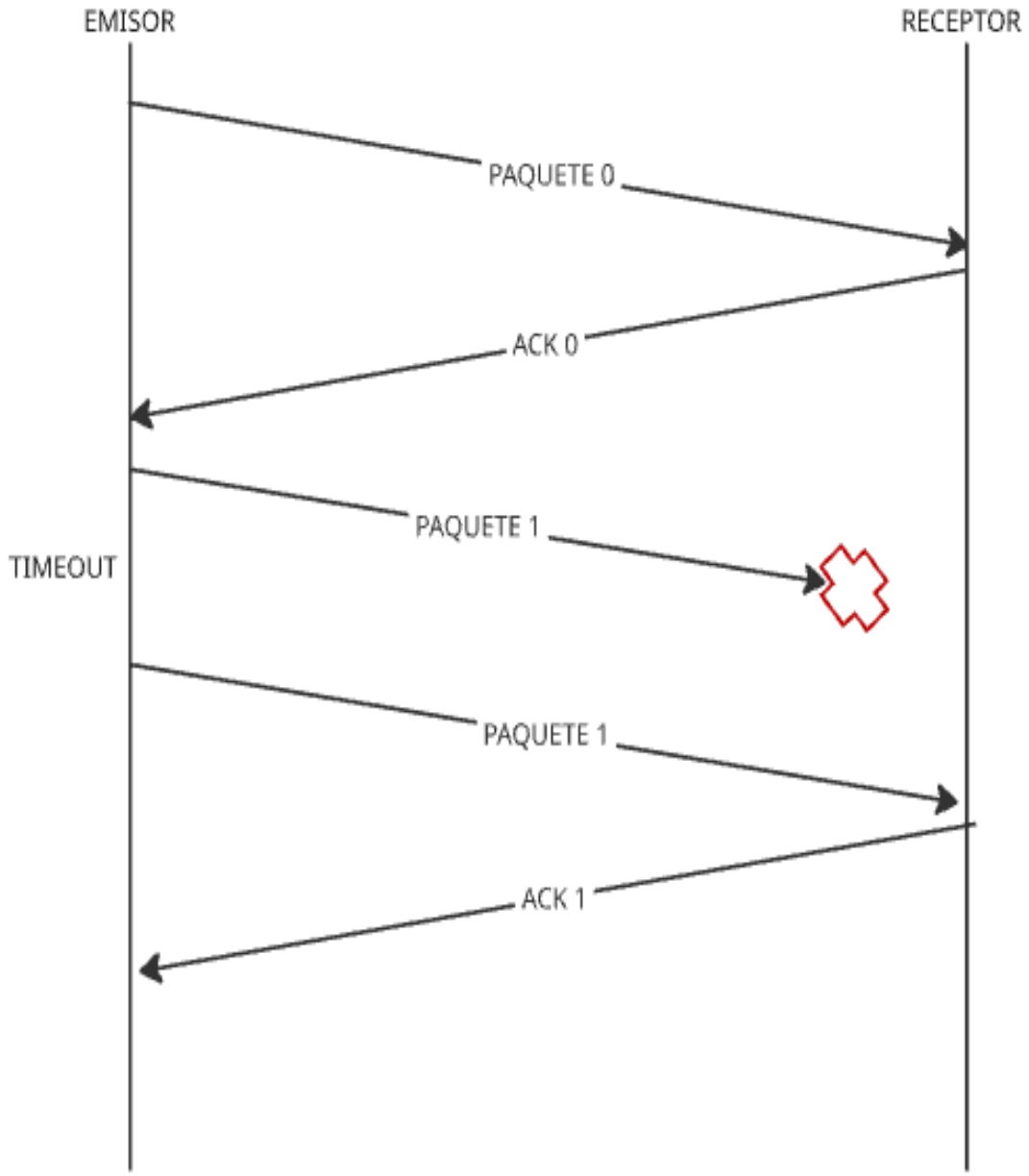
Si el ACK no es recibido dentro de un tiempo determinado, el emisor retransmite el paquete. Este comportamiento garantiza la entrega confiable de los datos, aunque puede limitar el rendimiento en conexiones de alta latencia.

La comunicación utiliza una secuencia alternada entre 0 y 1, que permite distinguir entre paquetes nuevos y retransmitidos. En caso de pérdida de paquetes o confirmaciones, se aplica un timeout fijo de 2 segundos, tras el cual el emisor reenvía el paquete pendiente.

En modo envío (pudiendo ser un UPLOAD estando del lado del cliente o un DOWNLOAD del lado del servidor) se lee el archivo en fragmentos de CHUNK_SIZE (que es el tamaño del payload de los paquetes que se envían). Los paquetes tienen la forma seq:is_last ya que se informa a la otra parte cuál es el número de secuencia correspondiente a ese paquetes (solo 0 o 1 en este caso) y el is_last para informar si es el último fragmento o no

El lado de la comunicación que está haciendo el envío espera un ACK específico antes de avanzar (el que corresponda al número de secuencia que envió previamente)

En modo recepción (pudiendo ser UPLOAD estando del lado del servidor o DOWNLOAD del lado del cliente) se valida el número de secuencia (que sea el que se esperaba recibir) y en caso de que sea correcto se escribe en el almacenamiento propio el fragmento del archivo que se está construyendo, y se responde con un ACK<seq> (donde seq es el que se recibió de la otra parte). Ante duplicados o pérdidas se reenvía el último ACK válido, y se tiene un tope de esperas antes de dar la sesión por caída (y no volver a reintentar)


EMISOR
RECEPTOR
PAQUETE 0
ACK 0
PAQUETE 1
TIMEOUT
PAQUETE 1
ACK 1

Go-Back-N

El protocolo Go-Back-N es una variante más avanzada del mecanismo de confiabilidad en comparación con Stop & Wait, ya que permite mantener hasta N paquetes en vuelo simultáneamente. Este enfoque implementa una ventana deslizante (sliding window) que se desplaza a medida que el emisor recibe confirmaciones (ACKs) por parte del receptor.

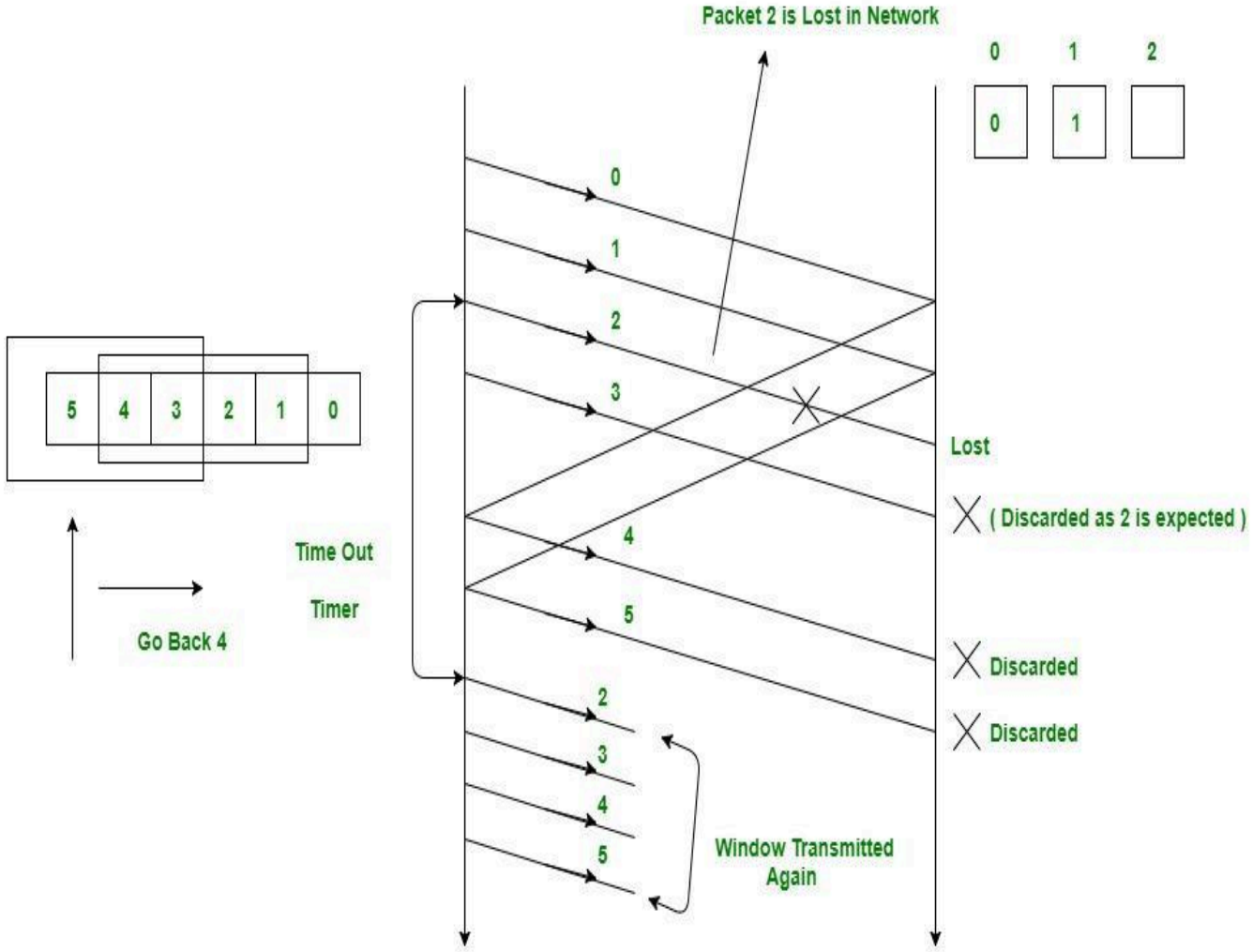
El emisor envía un conjunto de N paquetes consecutivos y activa un temporizador interno al transmitir el primero. A medida que llegan los ACKs, el número de secuencia confirmado indica cuál será el próximo paquete a enviar, permitiendo mover la ventana hacia adelante.

El timeout utilizado es adaptativo, ajustándose dinámicamente según las condiciones de la red. En caso de pérdida de un paquete o falta de confirmación, el emisor retransmite toda la ventana comenzando desde el primer paquete no reconocido, garantizando así la integridad y el orden de los datos transferidos.

En modo envío se divide al archivo en bloques de MSS (pudiendo el último de todos tener un tamaño menor) y se manda un paquete que contiene DATA<seq>:<is_last> seguido de la información que se está mandando en ese paquete. Se mantiene una ventana de transmisión de tamaño fijo en 8 paquetes y para cada paquete enviado se guarda el momento en el que fue transmitido

Cada ACK recibido actualiza la base de la ventana, además se calcula RTT estimado y desviación para ajustar dinámicamente el timeout mínimo

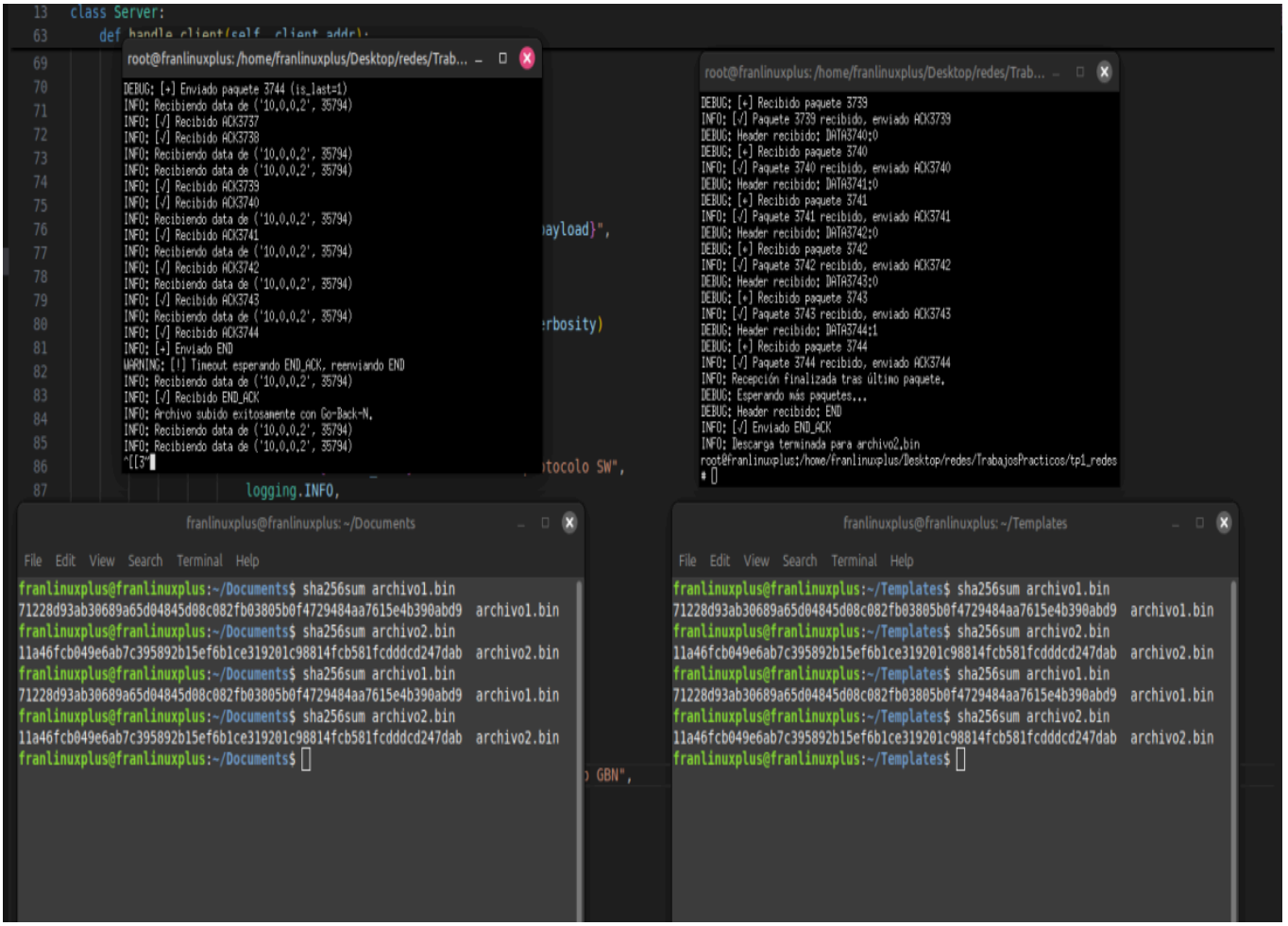
En modo recepción se acepta solo el expected_seq_number, y si este es correcto entonces se escribe el fragmento del archivo en el almacenamiento propio y se emite un ACK<seq>. En el último paquete se marca un is_last = 1 y se espera un END explícito de la otra parte, para finalizar con un END_ACK como respuesta (lo que libera recursos del servidor de forma ordenada)



Pruebas

Se realizaron operaciones de **Download y Upload** con archivos de **5 MB**, empleando como protocolo de recuperación Go-Back-N. El objetivo fue medir el comportamiento del protocolo bajo las condiciones de red utilizadas en los ensayos.

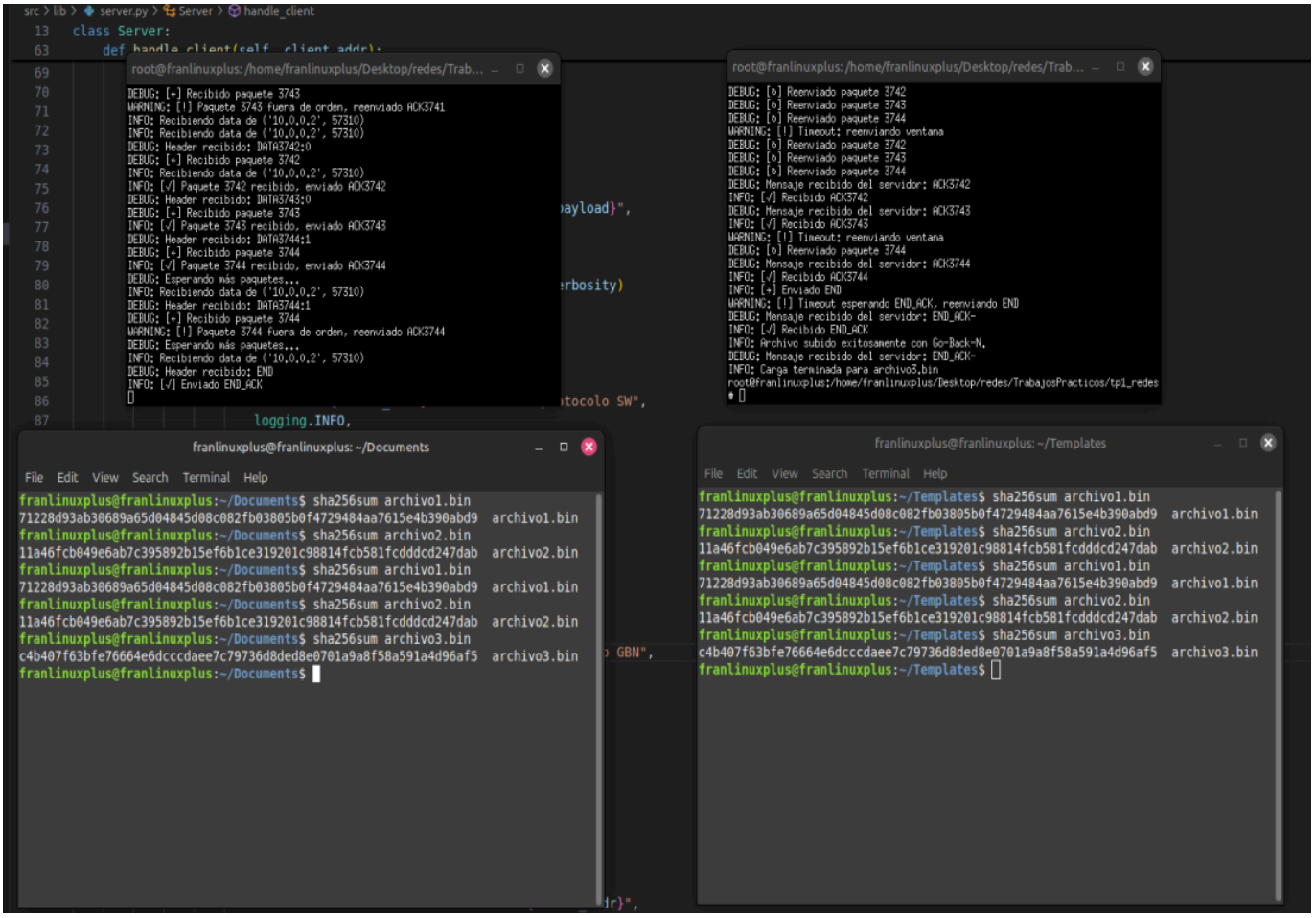
Operación	Tiempo total	% Pérdida de Paquetes
DOWNLOAD	54.04 segundos	10



En la imagen se muestra el resultado final de la operación de transferencia. Mediante la utilización del comando sha256sum, se verificó la integridad del archivo recibido, confirmando que el hash obtenido coincide con el del archivo original.

Esto demuestra la correcta ejecución del proceso de transferencia, sin alteraciones ni pérdidas de datos durante la transmisión.

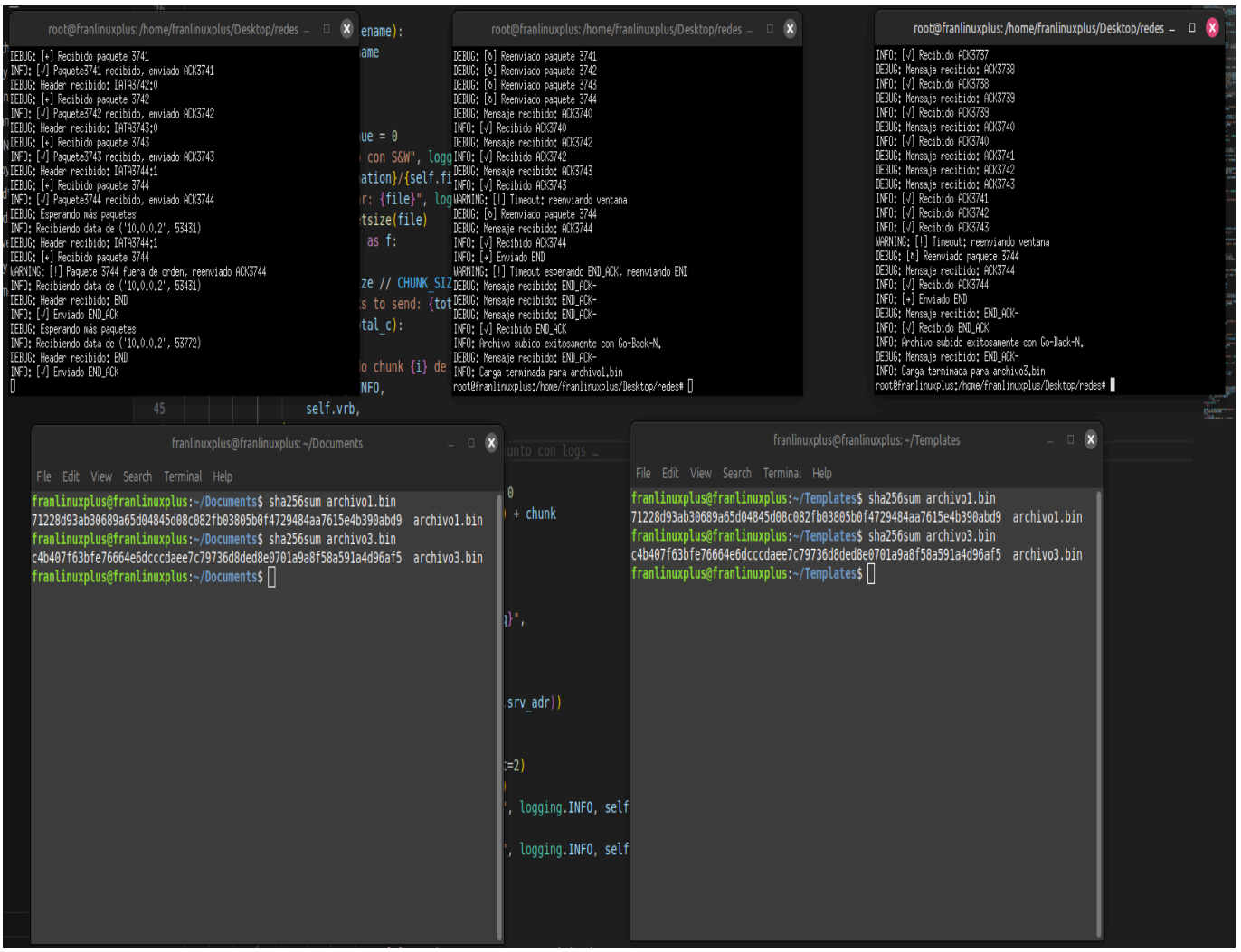
Operación	Tiempo total	% Pérdida de Paquetes
UPLOAD	52.84 segundos	10



Del mismo modo que en la operación DOWNLOAD, se empleó el comando sha256sum para verificar la integridad del archivo transferido. La coincidencia entre los valores hash del archivo original y el recibido confirma que la transferencia se completó correctamente, sin errores ni alteraciones en los datos.

Por otro lado también se hicieron pruebas para medir la capacidad del servidor de poder atender a varios clientes de forma concurrente sin que ello implique que la velocidad de transmisión empeore.

Operación	Tiempo total	% Pérdida de Paquetes
2 UPLOAD	52.15 segundos	10

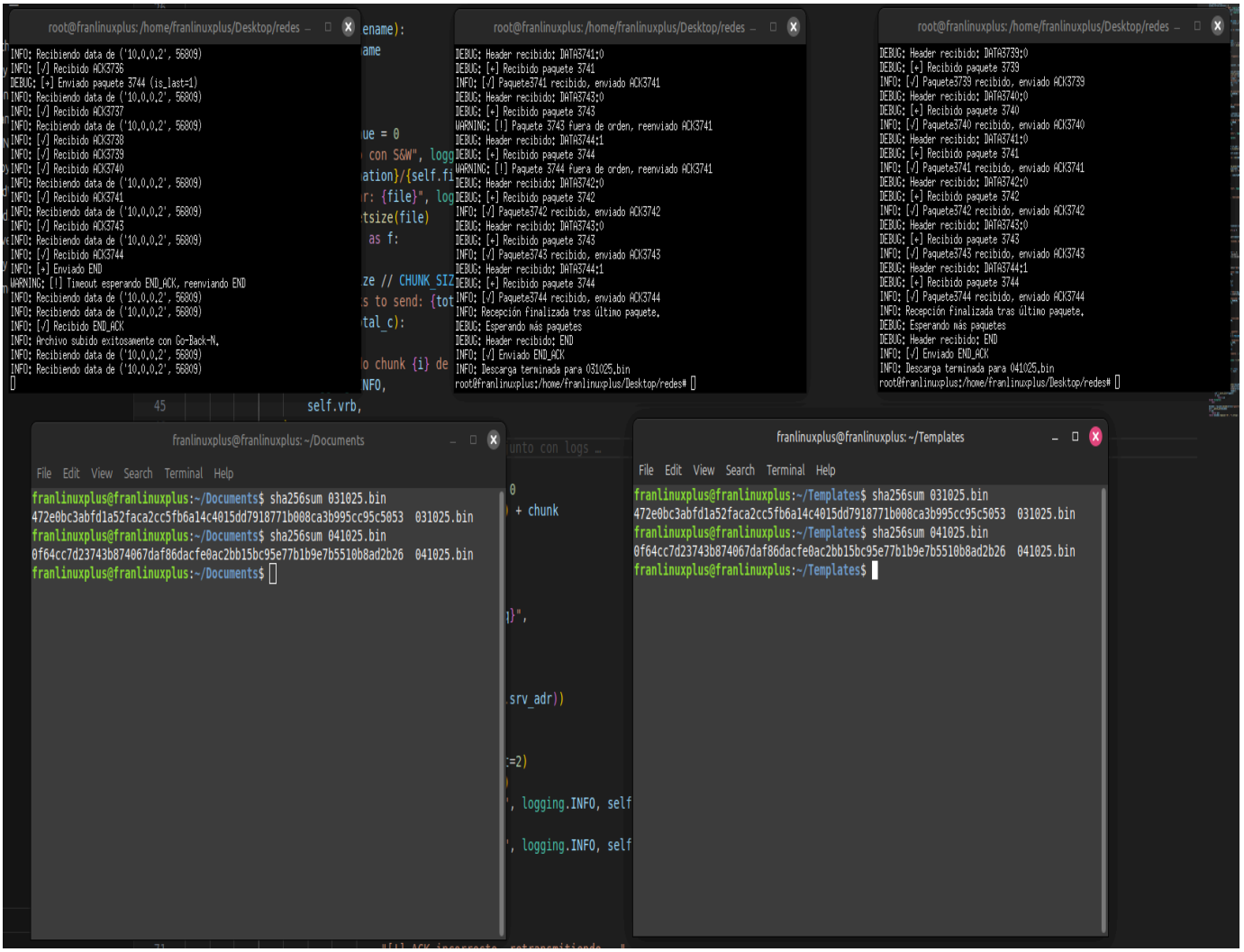


Como se puede ver en este caso un cliente envió un archivo1 al mismo tiempo que otro envió un archivo3, y ambos llegaron correctamente lo que es indicado porque el checksum coincide en ambos directorios (el original y a donde fueron enviados)

Esto demuestra que el servidor es capaz de soportar múltiples usuarios que suban archivos distintos

Si se hace una prueba similar pero en este caso descargando archivos

Operación	Tiempo total	% Pérdida de Paquetes
2 DOWNLOAD	64.92 segundos	10

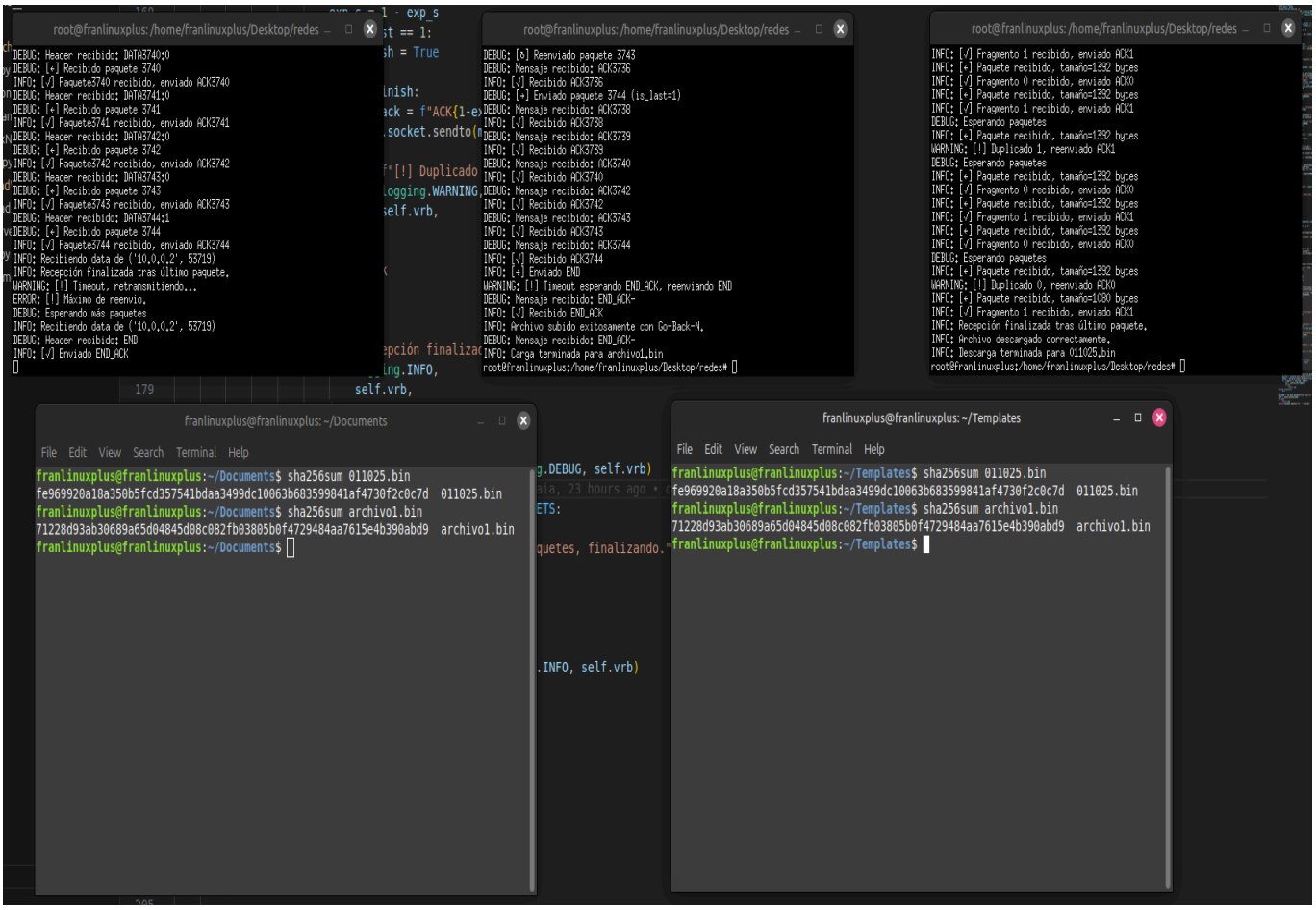


Se puede ver en imagen que en este caso los dos clientes descargaron otros dos archivos diferentes entre sí al mismo tiempo y en ambos casos el archivo se recibió correctamente (lo que se comprueba con que el checksum coincide en ambos directorios)

De esta forma se muestra que el servidor es capaz de soportar a múltiples usuarios que descarguen archivos diferentes de forma simultánea. Y en conjunto con lo anterior se muestra que si varios usuarios quieren realizar operaciones utilizando un mismo protocolo a la vez entonces el servidor puede responder a todos ellos y que no haya un empeoramiento significativo del tiempo total que se tarda en completar todas las operaciones

Por último además de probar la capacidad de concurrencia del servidor también se quiere saber si es capaz de atender a varios clientes que utilicen diferentes protocolos, ya que el servidor debe poder ser capaz de responder a clientes con diferentes condiciones de red y que utilicen diferentes protocolos debido a esto (además que se pueden querer transferir archivos de diferentes tamaños). En este sentido se hizo una prueba de nuevo con dos clientes para verificar esto

Operación	Tiempo total	% Pérdida de Paquetes
1 DOWNLOAD y 1 UPLOAD	59.64 segundos	10



En este caso se hizo una descarga de un archivo de 100K utilizando el protocolo Stop & Wait y una carga de un archivo de 5MB utilizando el protocolo Go-Back-N al mismo tiempo. Se puede ver que ambos archivos pudieron transferirse correctamente ya que el checksum coincide en los diferentes directorios

De esta forma se muestra que el servidor soporta a múltiples clientes que utilicen diferentes protocolos entre sí y puede atender a todos sin que eso signifique un empeoramiento significativo del tiempo total en que tarden todas las operaciones en completarse. Esto es un escenario más real donde no necesariamente todos los clientes utilicen el mismo protocolo ya que como se dijo anteriormente el usar uno u otro en cada caso puede depender del tamaño del archivo a transferir y de las condiciones de la red en la que se lo haga

Preguntas

1) Describa la arquitectura Cliente-Servidor.

La **arquitectura Cliente-Servidor** es un modelo de diseño de sistemas distribuidos en el que las funciones y responsabilidades se dividen en dos roles principales:

1. Cliente

- Es la parte del sistema que **realiza peticiones**.
- Suele estar en manos del usuario final (por ejemplo, un navegador web, una app de escritorio o móvil).
- Sus tareas principales son:
    - Enviar solicitudes al servidor (por ejemplo, pedir una página web, consultar datos, enviar información).
    - Mostrar los resultados al usuario de forma comprensible.

2. Servidor

- Es la parte que **responde a las peticiones** del cliente.
- Generalmente se ejecuta en máquinas más potentes o especializadas, ya que recibe, almacena y procesa todos los datos provenientes de todos los clientes
- Sus funciones principales son:
    - Procesar las solicitudes recibidas.
    - Acceder a bases de datos o recursos necesarios.
    - Enviar la respuesta de vuelta al cliente.

Características Clave

- **Comunicación a través de la red**: Cliente y servidor se conectan mediante protocolos de comunicación, como **HTTP/HTTPS** en la web.

- **Centralización de recursos**: El servidor concentra la lógica, la seguridad y el acceso a datos, mientras que el cliente solo actúa como interfaz.

- **Escalabilidad**: Es posible tener múltiples clientes conectados a un mismo servidor, o incluso varios servidores balanceando la carga.

- **Independencia**: Los clientes no necesitan saber cómo funcionan internamente los servidores, solo cómo interactuar con ellos mediante una interfaz o protocolo definido.

En el contexto de nuestra aplicación de transferencia de archivos (operaciones de subida y descarga), este modelo se implementa de la siguiente manera:

El servidor opera como un nodo central encargado de recibir y procesar las solicitudes provenientes de múltiples clientes, utilizando sockets basados en el protocolo UDP para la comunicación y un hilo independiente para cada solicitud (de manera de poder manejar varios clientes de forma concurrente). Esta estructura multihilo permite realizar múltiples transferencias de archivos de forma simultánea, sin bloquear la atención de nuevos clientes. Este servidor puede ejecutar dos operaciones principales: enviar un archivo solicitado (descarga) o recibir un archivo enviado por el cliente (subida)

El cliente es el componente que inicia la comunicación con el servidor para solicitar o enviar archivos. Al igual que el servidor puede ejecutar dos tipos de operación: subida de archivos (upload) al servidor en donde el cliente le envía partes del mismo en los paquetes y el servidor los va reconstruyendo hasta que lo termina guardando en su almacenamiento local, y descarga de archivos (download) desde el servidor donde el cliente solicita un archivo determinado y, al igual que el servidor, lo recibe de a partes en los paquetes hasta que lo reconstruye completamente. Para garantizar la entrega confiable de los paquetes sobre UDP se utiliza un protocolo de control de errores como Stop & Wait o Go-Back-N , los cuales aseguran la correcta recepción de datos por parte del servidor

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

La función de un protocolo de capa de aplicación es **definir las reglas y convenciones que permiten a los procesos de aplicación de distintos sistemas comunicarse entre sí de manera correcta**. En otras palabras, la capa de aplicación se centra en **qué datos se intercambian, cómo se estructuran y qué acciones deben realizarse con ellos**.

Un protocolo de aplicación establece:

- **Formato de los mensajes**: cómo se representan los datos, encabezados, campos y estructuras que deben respetar tanto cliente como servidor.

- **Semántica de los mensajes**: qué significa cada tipo de mensaje (por ejemplo, una petición de descarga, una respuesta exitosa, un error, etc.).

- **Reglas de interacción**: el orden en que los mensajes pueden enviarse, cómo deben responder los participantes y qué hacer en caso de errores o condiciones excepcionales.

Gracias a estos protocolos, es posible lograr la **comunicación entre aplicaciones** que pueden estar desarrolladas en diferentes plataformas o lenguajes, pero que comparten un conjunto común de reglas de comunicación. Ejemplos conocidos de protocolos de capa de aplicación son **HTTP** (para la web) o **SMTP** (para correo electrónico).

En el contexto de este trabajo, el protocolo de capa de aplicación diseñado tiene la función de **carga (UPLOAD) y descarga (DOWNLOAD) de archivos entre cliente y servidor**, asegurando que los datos se transfieran de manera confiable sobre un medio de transporte no confiable como **UDP**, mediante mecanismos de control de errores y recuperación.

3) Detalle el protocolo de aplicación desarrollado en este trabajo

El protocolo de aplicación desarrollado define las reglas de comunicación entre cliente y servidor dentro de una arquitectura distribuida basada en datagramas. Su función principal es permitir la **transferencia confiable de archivos** entre ambas partes, implementando mecanismos de control y verificación a nivel de aplicación para compensar la falta de confiabilidad del medio de transporte subyacente.

La interacción comienza cuando el cliente se conecta al servidor enviando un mensaje de inicialización que contiene información sobre la acción solicitada (por ejemplo, upload o download de un archivo) y el nombre del archivo involucrado. El servidor, al recibir este mensaje, interpreta la solicitud, envía una confirmación (ACK) y crea un hilo dedicado para atender al cliente de manera independiente. De este modo, el servidor puede gestionar múltiples clientes simultáneamente.

Los mensajes transmitidos están estructurados mediante datagramas que combinan un encabezado (header) y una sección de datos (payload). El encabezado contiene información esencial como el número de secuencia, indicadores de control (por ejemplo, si es el último paquete) y campos de verificación. Gracias a esta estructura, el protocolo puede mantener el orden correcto de los paquetes y detectar pérdidas o duplicados.

Durante la transferencia de datos, cada paquete enviado por el cliente debe ser confirmado por el servidor mediante un mensaje de reconocimiento (ACK). Si el cliente no recibe la confirmación dentro de un tiempo límite, el paquete es retransmitido. Este mecanismo garantiza que todos los fragmentos del archivo lleguen correctamente al destino, incluso en presencia de errores o pérdidas en la red.

El servidor ensambla los fragmentos recibidos en un archivo completo, escribiéndolos secuencialmente en disco según su número de secuencia. Una vez completada la transferencia, se intercambian mensajes de finalización (por ejemplo, “END” y “END_ACK”) que indican el cierre correcto de la sesión y liberan los recursos asociados.

El protocolo también implementa un sistema de **timeout dinámico**, calculado a partir de estimaciones del **retardo de ida y vuelta (RTT)**. Esto permite adaptar los tiempos de

retransmisión al comportamiento real de la red, mejorando la eficiencia del proceso y reduciendo la congestión.

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?

La **capa de transporte** en el modelo TCP/IP es la encargada de proporcionar comunicación extremo a extremo entre procesos que se ejecutan en distintos equipos de la red. Dentro de esta capa se destacan dos protocolos principales: **TCP (Transmission Control Protocol)** y **UDP (User Datagram Protocol)**.

TCP (Transmission Control Protocol)

- **Servicios que provee:**
    - Comunicación confiable mediante el mecanismo de **control de errores** y **retransmisión** de paquetes perdidos.
    - **Control de flujo** para evitar que un emisor sobrecargue al receptor.
    - **Control de congestión** para adaptarse a la capacidad de la red.
    - Transferencia de datos en **secuencia**, garantizando que los mensajes lleguen en el mismo orden en que fueron enviados.
- **Características:**
    - Antes de transmitir datos se establece una conexión entre emisor y receptor (triple handshake).
    - Ofrece un **canal de comunicación fiable y continuo**.
    - Header grande (20 bytes mínimo).
- **Cuándo usarlo:**
    - En aplicaciones donde la **confiabilidad** y la **integridad de los datos** son prioritarias, como en la **web (HTTP/HTTPS)**, **correo electrónico (SMTP/IMAP/POP3)** o **transferencia de archivos (FTP)**.

UDP (User Datagram Protocol)

- **Servicios que provee:**

    - Comunicación simple y rápida mediante el envío de **datagramas no confiables**.
    - Mínimo control de errores (checksum básico en cabecera).
- **Características:**
    - No requiere establecer ni mantener una conexión previa.
    - No garantiza entrega, orden ni ausencia de duplicados.
    - Header pequeño (8 bytes).
    - Mayor rapidez y menor consumo de recursos comparado con TCP.
- **Cuándo usarlo:**
    - En aplicaciones donde **la velocidad y la eficiencia** son más importantes que la confiabilidad, como en **streaming de audio o video en tiempo real**, **DNS**, **juegos en línea** y protocolos que implementan su propia confiabilidad (como en este trabajo, mediante Stop & Wait y Go-Back-N)

Dificultades encontradas

Durante el desarrollo del proyecto nos enfrentamos a diversas dificultades que requirieron iteraciones constantes y replanteos en nuestra solución. A continuación detallamos los principales desafíos:

- Diseño inicial de la aplicación: una de las primeras dificultades fue definir una arquitectura modular y escalable que permitiera implementar y comparar distintos protocolos de transferencia de archivos. Determinar cómo estructurar las entidades principales (cliente y servidor) y establecer responsabilidades claras para cada componente requirió tiempo y pruebas.
- Sincronización entre cliente y servidor: uno de los problemas más desafiantes fue lograr una correcta sincronización entre ambas partes durante la transferencia de datos, ya que cualquier desfasaje en los números de secuencia y ACKs entre cliente y servidor podía generar duplicación de datos, pérdidas o incluso dejar la comunicación en un estado inconsistente.
- Lograr que la comunicación se cierre correctamente, lo que requirió diferentes tipos de soluciones propuestas y encontrarnos con los fallos de cada una hasta que logramos una configuración en la que ya funciona.

- El manejo de la concurrencia cuando hay más de un cliente operando simultáneamente, que nuevos clientes puedan sumarse a hacer uploads ó downloads cuando ya hay clientes que los estaban haciendo antes (y que estos últimos continúen normalmente).

Conclusión

El desarrollo de esta aplicación cliente-servidor para la transferencia de archivos ha sido un proyecto desafiante, que nos permitió profundizar en los fundamentos de redes y protocolos de comunicación, tanto desde un enfoque teórico como práctico. A lo largo del proyecto exploramos las principales características necesarias para lograr una transferencia de archivos confiable. El protocolo Stop & Wait se destacó por su simplicidad y facilidad de implementación, siendo adecuado para situaciones de bajo tráfico pero ya cuando este aumenta sus tiempos se alargan bastante por la cantidad de retransmisiones necesarias que hay que hacer en los paquetes. Por otro lado Go-Back-N demostró ser una solución más robusta y eficiente gracias al uso de ventanas deslizantes que permite mantener una mayor cantidad de paquetes en vuelo en la comunicación (y con esto reduciendo los tiempos considerablemente). Aunque esta eficiencia adicional vino acompañada de una mayor complejidad en su diseño e implementación debido a la gestión de múltiples temporizadores, buffers y reordenamiento de paquetes.

Los resultados obtenidos no solo confirmaron la confiabilidad de ambos protocolos sino que también ofrecieron evidencia concreta sobre sus diferencias en cuanto a rendimiento y escalabilidad. En contextos reales, donde las condiciones de red pueden variar significativamente, Go-Back-N se posiciona como una solución más robusta y escalable, aunque Stop & Wait no debe ser descartado en situaciones controladas o de baja complejidad.