Trabajo Práctico
File Transfer

Redes
Segundo cuatrimestre de 2025

ALUMNO	PADRON	CORREO
BASSO, Catalina	108564	cbasso@fi.uba.ar
DE BENEDETTO, Celeste	108082	cdebenedetto@fi.uba.ar
NATALE, Nicolas	108590	nnatale@fi.uba.ar
ROMAN, Lisandro	107274	liroman@fi.uba.ar
SALLUZI, Luca	108088	lsalluzzi@fi.uba.ar

Índice

1. Introducción

La capa de transporte de red ofrece protocolos como TCP, que garantiza una entrega de datos fiable, y UDP, que al no hacerlo, resulta inviable para aplicaciones como la transferencia de archivos que exigen integridad total de los datos.

Este trabajo práctico aborda ese desafío mediante el diseño de una aplicación cliente-servidor que construye un servicio de transferencia de archivos fiable sobre UDP. Para ello, se implementaron y compararon dos protocolos de Transferencia de Datos Confiable: el simple Stop and Wait y el más eficiente Selective Repeat, que utiliza ventanas deslizantes.

Para validar la robustez de ambas implementaciones y analizar su rendimiento, la aplicación fue desplegada en una red virtual con pérdida de paquetes simulada a través de la herramienta Mininet. El presente informe detalla el diseño, la implementación y los resultados comparativos de estos protocolos.

2. Hipótesis y suposiciones realizadas

Para el desarrollo de las implementaciones se asumió que, una vez finalizada correctamente una transferencia (ya sea en el envío o la recepción de un archivo), el cliente se desconecta del servidor y no realiza una nueva operación de forma inmediata. Es decir, cada ejecución contempla únicamente una solicitud de transferencia por conexión.

Asimismo, se considera que el servidor maneja cada solicitud de cliente de forma completamente independiente. Una vez concluida una transferencia, el servidor libera todos los recursos asociados a la conexión correspondiente, sin mantener estado alguno para futuras interacciones con el mismo cliente.

Por otro lado, el tamaño de los buffers y de las ventanas de envío se encuentra fijado mediante constantes predefinidas, lo cual simplifica el diseño y permite un control más directo de los parámetros involucrados en la transmisión.

Finalmente, en caso de producirse un error durante la transferencia —por cualquier motivo— no se contempla la posibilidad de reanudar la operación desde el punto de interrupción. Ante este tipo de situaciones, la transferencia debe reiniciarse completamente desde el principio.

3. Implementación

3.1. Protocolo de comunicación

Para cumplir con los objetivos del trabajo práctico, se diseñó e implementó un protocolo de capa de aplicación personalizado que opera sobre UDP. El propósito de este protocolo es orquestar la transferencia de archivos entre el cliente y el servidor, permitiendo las operaciones de *UPLOAD* y *DOWNLOAD*, y negociar qué mecanismo de transferencia de datos confiable (RDT) se utilizará: **Stop-and-Wait** o **Selective Repeat**.

El protocolo se puede dividir en tres fases bien definidas:

1. Handshake (Saludo Inicial)
2. Negociación y Asignación de Canal
3. Transferencia de Datos Confiable

A continuación, se detalla cada fase.

3.1.1. Fases del Protocolo

1. Fase de Handshake (Saludo Inicial) La comunicación siempre es iniciada por el cliente, que envía un único paquete UDP al puerto principal del servidor. Este paquete inicial, o "saludo", informa al servidor sobre la intención del cliente. El formato del mensaje varía según la operación:

- **Para UPLOAD**: El cliente anuncia el archivo que desea subir, su tamaño y el protocolo RDT a utilizar. El formato es una única cadena de texto, con las partes separadas por dos puntos (:).

  `"UPLOAD_CLIENT:<protocol>:<filename>:<filesize>"`

- **Para DOWNLOAD**: De manera similar, el cliente solicita un archivo, especificando su nombre y el protocolo RDT.

  `"DOWNLOAD_CLIENT:<protocol>:<filename>"`

Este primer contacto se realiza con el socket principal del servidor, que se mantiene escuchando exclusivamente por estos mensajes de saludo.

2. Fase de Negociación y Asignación de Canal Esta fase es crucial para permitir la concurrencia. Una vez que el socket principal del servidor recibe un saludo válido, realiza dos acciones clave:

1. **Crea un Hilo Dedicado**: El servidor instancia un nuevo hilo (`threading.Thread`) que se encargará exclusivamente de la transferencia con ese cliente.
2. **Abre un Nuevo Socket**: Dentro del nuevo hilo, el servidor crea un socket UDP en un puerto temporal asignado por el sistema operativo.

Inmediatamente después, el servidor responde al cliente, comunicándole el nuevo puerto que debe usar para la transferencia.

- **Respuesta para UPLOAD**:

  `"UPLOAD_OK:<new_port>"`

- **Respuesta para DOWNLOAD**: Si el archivo existe, responde con el nuevo puerto y el tamaño del archivo.

  `"DOWNLOAD_OK:<new_port>:<filesize>"`

  Si no existe, se envía un mensaje de error: `ERROR:FileNotFound`.

Al recibir esta respuesta, el cliente reconfigura su destino al nuevo puerto, y toda la comunicación posterior se realiza a través de este canal dedicado.

3. Fase de Transferencia de Datos Confiable (RDT) Con el canal dedicado ya establecido, el control pasa al manejador del protocolo RDT seleccionado. En esta fase se transfiere el archivo de forma confiable.

- **Paquetes de Datos**:

  `b"<seq_num>:<chunk>"`

- **Paquetes de Confirmación (ACKs)**:

  `b"ACK:<seq_num>"`

Toda la lógica de timeouts, retransmisiones y ventanas deslizantes ocurre durante esta fase, siguiendo las reglas del protocolo RDT elegido.

3.1.2. Protocolo Stop & Wait

Para la primera versión de la transferencia de datos confiable, se implementó el protocolo **Stop & Wait**. Este mecanismo es el más simple de los protocolos RDT, caracterizado por permitir un único paquete "en vuelo" a la vez. El emisor no puede enviar un nuevo paquete hasta no haber recibido la confirmación (ACK) del anterior.

Lógica del Emisor: El corazón de la implementación del emisor se encuentra en el método `_send_packet_reliable`. Al enviar cada paquete, el emisor inicia un temporizador con un timeout adaptativo. Este timeout se calcula dinámicamente a partir del RTT (Round-Trip Time) estimado, ajustándose a las condiciones de la red. Si el ACK esperado no llega antes de que el temporizador expire, el emisor asume que el paquete o su ACK se perdieron y lo retransmite. Este proceso se repite hasta un número máximo de reintentos (`MAX_RETRIES`). Para detectar duplicados, se utiliza un número de secuencia que alterna entre 0 y 1 para cada paquete consecutivo.

Lógica del Receptor: El receptor, por su parte, mantiene una variable de estado que indica el número de secuencia del paquete que está esperando (`seq_expected`). Al recibir un paquete, verifica:

- Si el número de secuencia es el esperado, lo acepta, escribe el contenido en el archivo, envía el ACK correspondiente y actualiza su estado para esperar el siguiente número de secuencia (alternando entre 0 y 1).
- Si el número de secuencia no es el esperado, significa que es un paquete duplicado. En este caso, el receptor descarta los datos pero vuelve a enviar el ACK del último paquete correcto que recibió, para asegurar que el emisor pueda avanzar.

3.1.3. Protocolo Selective Repeat

Como alternativa de alto rendimiento, se implementó el protocolo **Selective Repeat**. A diferencia de Stop & Wait, este mecanismo mejora drásticamente la eficiencia al permitir que múltiples paquetes estén en tránsito simultáneamente, utilizando una **ventana deslizante**.

Ventana Deslizante y Envío: El emisor utiliza una ventana de envío de tamaño fijo (`WINDOW_SIZE`). Este puede enviar todos los paquetes que quepan en la ventana sin esperar confirmaciones individuales. Un diccionario, `pkts`, almacena los paquetes que han sido enviados pero aún no confirmados, junto con su tiempo de envío para el control de timeouts.

Recepción, Buffer y ACKs Selectivos: La clave del receptor en Selective Repeat es que puede recibir y almacenar paquetes fuera de orden, siempre y cuando pertenezcan a su ventana de recepción actual.

- Cada paquete válido que llega (dentro de la ventana) es confirmado individualmente con su propio ACK. Esto se conoce como **ACK selectivo**.
- Si un paquete llega fuera de orden (p. ej., llega el paquete 5 antes que el 4), es almacenado en un buffer temporal (`received_pkts`).
- La base de la ventana de recepción (`base_num`) solo avanza cuando se recibe el paquete esperado en orden. Al hacerlo, el receptor escribe en el archivo ese paquete y todos los paquetes consecutivos que ya se encontraban en el buffer.

Retransmisión Selectiva: El emisor mantiene un temporizador individual para cada paquete en vuelo. Si un temporizador expira, el emisor asume que ESE paquete en particular se perdió y retransmite **únicamente ese paquete**, no toda la ventana.

4. Pruebas

4.1. Stop and Wait

los resultados de las ejecuciones de prueba (capturas de ejecución de cliente y logs del servidor).

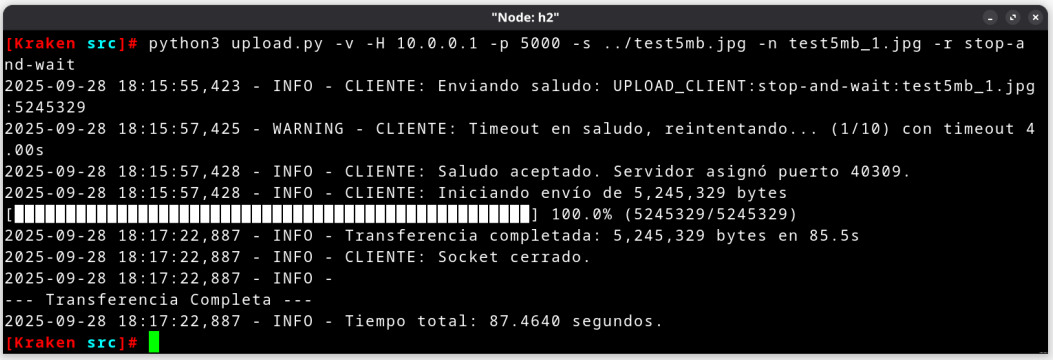

Figura 1: Client upload Stop and Wait

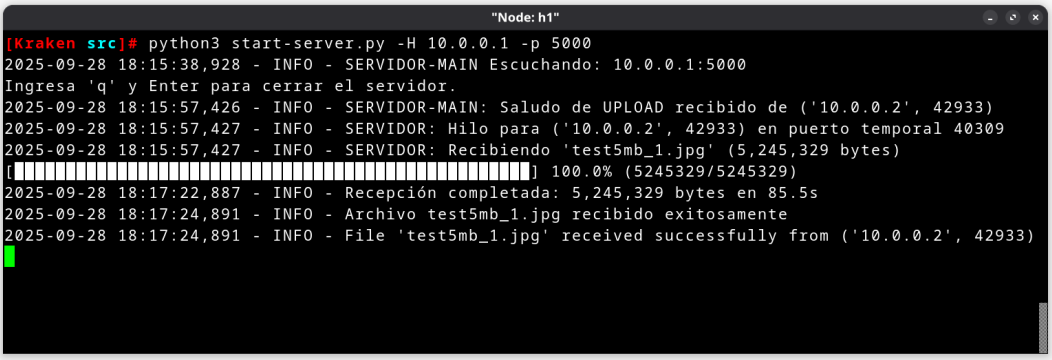

Figura 2: Server upload Stop and Wait

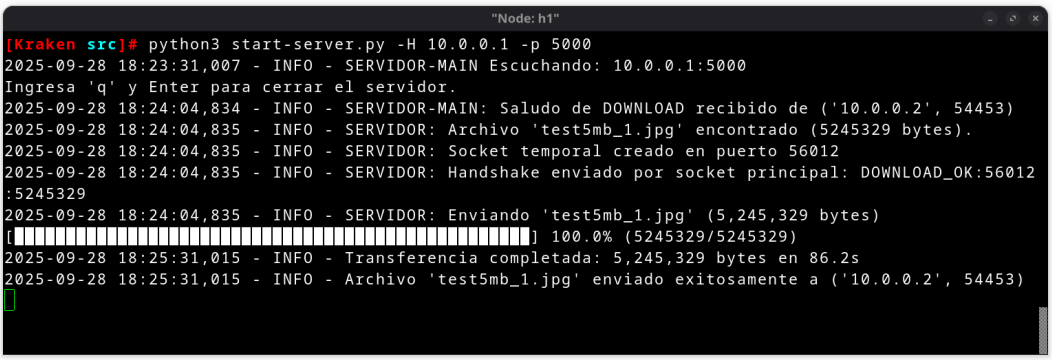

Figura 3: Server download Stop and Wait

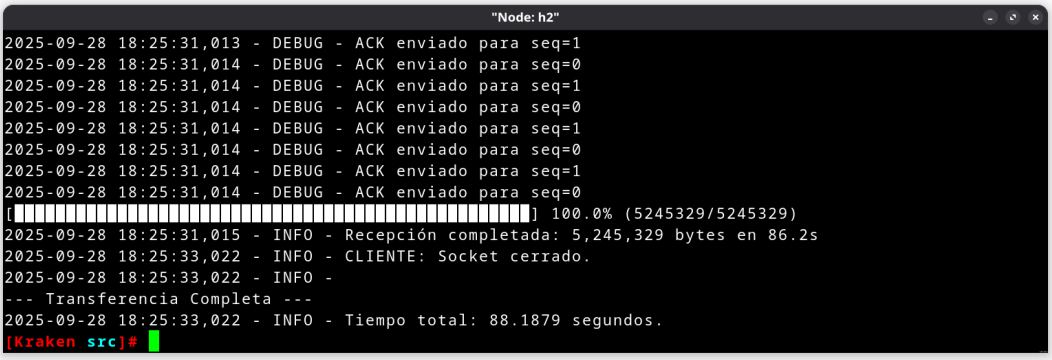

Figura 4: Client download Stop and Wait

4.2. Selective Repeat

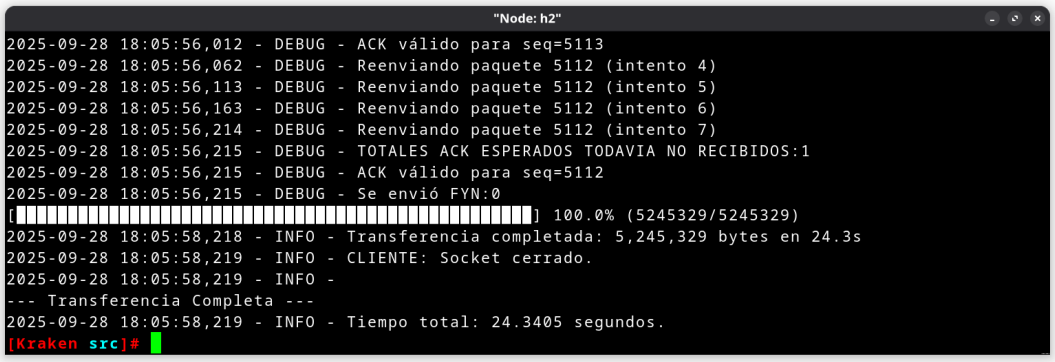

Figura 5: Client upload Selective Repeat

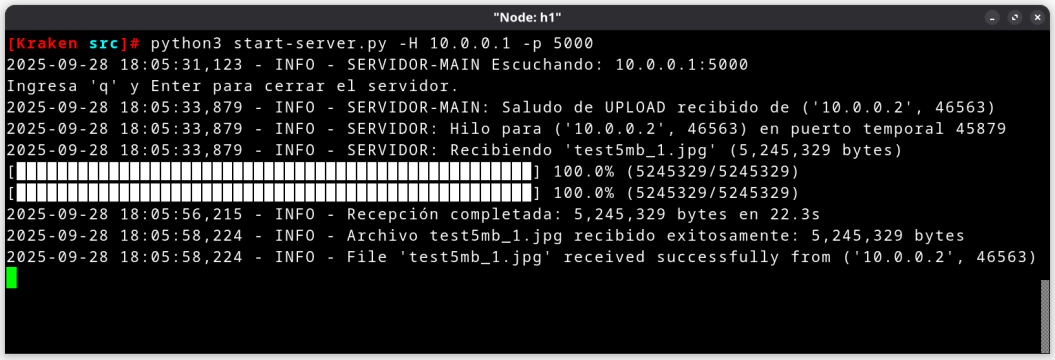

Figura 6: Server upload Selective Repeat

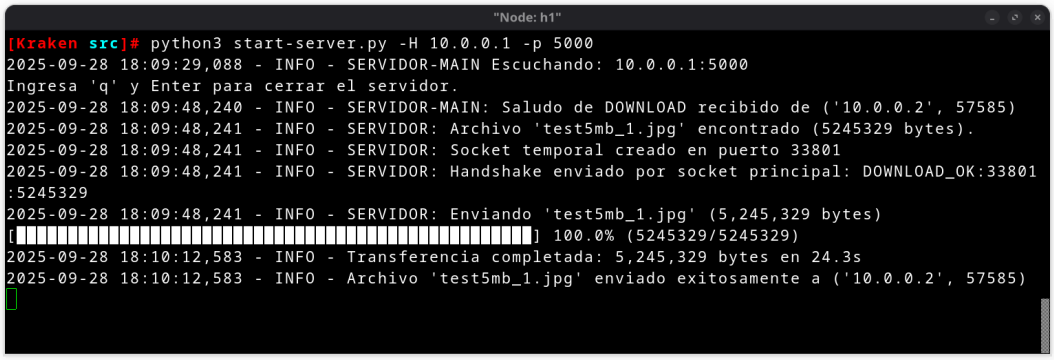

Figura 7: Server download Selective Repeat

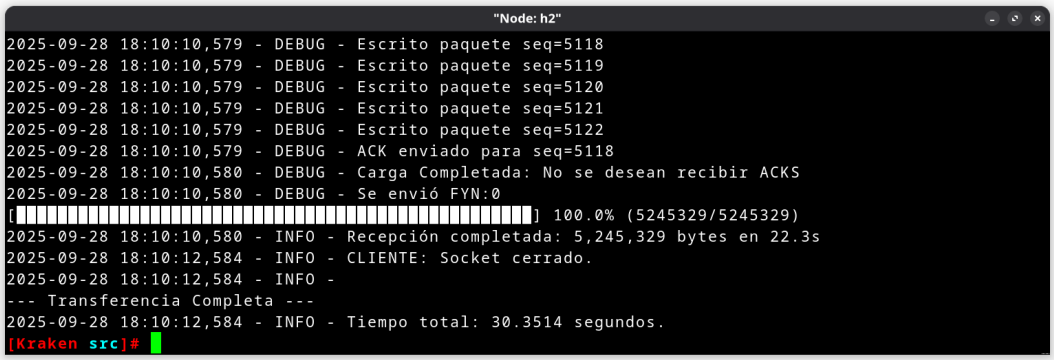

Figura 8: Client download Selective Repeat

5. Preguntas a responder

5.1. Describa la arquitectura Cliente-Servidor.

En la arquitectura cliente-servidor, siempre hay un anfitrión (host) siempre activo llamado servidor, que atiende las solicitudes de muchos otros anfitriones, llamados clientes. Las características clave de este modelo son:

- **Servidor con Dirección Fija:** El servidor tiene una dirección IP fija y conocida. Los clientes siempre saben a dónde enviar sus solicitudes.
- **Centro de Datos:** Un centro de datos, con su capacidad de escalar, puede albergar un único servidor que atiende a una enorme cantidad de clientes, lo que lo hace una arquitectura muy potente.
- **Comunicación Centralizada:** Los clientes no se comunican directamente entre sí; toda la comunicación pasa a través del servidor.

A la hora de cerrar las conexiones, el cliente cierra su conexión una vez completada su operación y termina su proceso de ejecución. No mantiene conexiones ni reutiliza la sesión para operaciones adicionales o futuras.

El servidor, por su parte, libera los recursos asociados a esa sesión específica (buffers, file descriptors, etc.) y queda disponible para atender nuevas solicitudes de otros clientes. Además, no conserva información sobre clientes previos entre sesiones. Cada nueva conexión se trata como una solicitud completamente independiente, sin historial ni contexto de interacciones anteriores.

5.2. 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 las aplicaciones que se ejecutan en distintos sistemas extremos (hosts) se comunican a través de la red.

Un protocolo de aplicación especifica:

- El tipo de mensajes que se intercambian (por ejemplo: solicitudes o respuestas).
- La sintaxis de esos mensajes (cómo están estructurados los campos de datos).
- La semántica (qué significan los campos de la información en un mensaje).
- Las reglas sobre cuándo y cómo los procesos envían y responden a los mensajes.

En resumen, su función es proporcionar las reglas que permiten que dos procesos de aplicación en extremos distintos puedan interoperar correctamente sobre la red.

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

El trabajo implementa un protocolo de capa de aplicación propio sobre UDP para transferir archivos. Este protocolo soporta operaciones de `UPLOAD` y `DOWNLOAD`, y permite negociar qué mecanismo de transferencia de datos confiable (RDT) se utilizará: *Stop and Wait* o *Selective Repeat*.

Handshake. El cliente envía un mensaje al servidor indicando si desea subir o bajar un archivo, el nombre, el tamaño y el protocolo RDT a utilizar.

Negociación y asignación de canal. El servidor abre un *socket* dedicado en un puerto temporal y le responde al cliente con ese nuevo puerto para continuar la transferencia.

Transferencia de datos. Una vez establecido el canal dedicado, se transfieren los datos usando el protocolo RDT elegido. Los datos se mandan en paquetes con número de secuencia y se confirman con `ACK`s.

5.4. 5.4. ¿Cuáles son sus características? ¿Cuándo es apropiado utilizar cada uno?

Stop and Wait. Es el protocolo más simple. Solo permite un paquete en vuelo a la vez y el emisor espera un `ACK` antes de mandar el siguiente. Es apropiado en redes con baja pérdida de paquetes y bajo retardo, donde la simplicidad prima sobre la eficiencia. Es útil para entornos de prueba y transferencias pequeñas.

Selective Repeat. Utiliza ventanas deslizantes, permite múltiples paquetes en tránsito, admite recepción fuera de orden, almacena el *buffer* hasta poder reordenar y envía `ACK`s individuales. Cada paquete tiene su temporizador y solo se retransmiten los que realmente se pierden. Es apropiado en redes con mayor latencia o pérdidas frecuentes de paquetes, ya que aprovecha mejor el ancho de banda y es más eficiente en transferencias grandes.

6. Dificultades encontradas

- **Manejo de pérdida de paquetes y sincronización**: Sin lugar a dudas, la mayor dificultad del trabajo fue —como era de esperar— el agregado de robustez para mitigar la pérdida de paquetes. Implementar ambos protocolos en un escenario ideal fue una tarea relativamente sencilla; sin embargo, al incorporar mecanismos de contención ante fallos, la complejidad aumentó considerablemente.

  La sincronización de los distintos `ACKs` resultó fundamental, ya que fue necesario establecer *timeouts* precisos para que las partes involucradas reaccionaran de manera ordenada. De esta forma, ante una eventual pérdida en la comunicación, el sistema podía recuperarse rápidamente sin generar demoras excesivas ni desperdicio de recursos en el cliente o el servidor.

- **Asignación de canal en la fase de negociación**: Otra dificultad relevante fue la asignación del canal de comunicación durante la fase de negociación entre cliente y servidor. Fue necesario orquestar cuidadosamente el envío y la recepción de la información para evitar que el cliente perdiera la conexión con el servidor, incluso ante la posible pérdida de algunos de los paquetes involucrados en esta etapa inicial.

7. Conclusión