Facultad de Ingeniería de la Universidad de Buenos Aires

Redes (TA048) Curso 02 Alvarez Hamelin

Trabajo Práctico N°1: File Transfer

Fecha de Entrega: 8 de Mayo de 2025

Landi, Agustina	107850
Di Nucci, Tomás	109628
Suarez Pino, Imanol	107746
Solari Vazquez, Federico	106895
Slepowron Majowiecki, María Mercedes	109454

1. Introducción

Este trabajo práctico tiene como objetivo la implementación de una aplicación de red para la transferencia de archivos siguiendo una arquitectura cliente servidor.

El desarrollo de la aplicación se llevó a cabo utilizando el protocolo UDP. Para garantizar un *Reliable Data Transfer*, se implementaron versiones de los protocolos Stop and Wait y Selective Repeat con el fin de contar con mecanismos para el control de errores y reenvío de paquetes.

Se tiene como objetivo comprender y profundizar en las herramientas necesarias para que el protocolo de transferencia de archivos sea RDT, y además comparar la performance de ambos, *Stop and Wait* y *Selective Repeat* en diferentes condiciones de la red.

2. Hipótesis y Suposiciones Realizadas

Al momento de desarrollar la aplicación propuesta, la selección del protocolo de la capa de transporte tiene un gran impacto no solo en el rendimiento del sistema, sino también en la velocidad, seguridad y fiabilidad que tiene el mismo para transportar la información y asegurar que esta llegue de manera íntegra, como lo deseamos. Realizaremos las siguientes suposiciones a la hora de implementar el sistema:

- Si tanto el servidor como el cliente trabajan en un entorno de localhost, no debería presentarse la pérdida de paquetes. Esto se debe a que, no estamos transportando los datos e información a través de la capa de redes; el envío se hace entre procesos, a través de sockets lo cual tiene un márgen de pérdida de información casi nulo. La simulación de pérdida se realizará corriendo el sistema a través de Mininet, para simular una situación más real.
- Se asume que los paquetes no vienen corrompidos, porque el protocolo UDP ya se asegura de realizar esta validación particular.
- Se trabajó con archivos de hasta 5MB como máximo, conforme al enunciado; por ende el tamaño de ventana, el número de secuencia y los buffers fueron dimensionados pensando en este escenario.
- Cada operación `UPLOAD` o `DOWNLOAD` es manejada entre el servidor (puede atender múltiples clientes concurrentemente pero cada socket atiende a una única operación a la vez) y el cliente.

3. Implementación

La aplicación desarrollada implementa los protocolos UDP y TCP embebidos en una arquitectura estándar de tipo cliente-servidor.

3.1. Servidor

La aplicación desarrollada cuenta con una arquitectura cliente - servidor, donde múltiples clientes se comunican con un único servidor para solicitar servicios de transferencia de archivos.

El servidor se inicia creando un socket UDP, en estado de escucha permanente mientras espera recibir mensajes entrantes de cualquier cliente. Cada mensaje recibido contiene sus datos y la dirección y puerto del cliente que lo envió; permitiendo así, identificar de forma única a cada conexión lógica.

Cuando llega un mensaje de un cliente nuevo, es decir un puerto no registrado, el servidor crea una cola de mensajes específica para ese cliente. Ésta actúa como canal de comunicación entre el hilo principal del servidor que recibe todos los mensajes y un hilo secundario que se encarga de manejar la lógica de comunicación con ese cliente. Para esto, se lanza una nueva instancia de Thread que ejecuta el método *handle_client_message*, asegurando que cada cliente sea atendido de forma independiente.

De esta manera el servidor atiende clientes de forma concurrente. Mientras el hilo principal sigue recibiendo mensajes de los distintos clientes, los hilos secundarios trabajan en paralelo, procesando los mensajes específicos de cada cliente, ejecutando el handshake, el protocolo de transferencia de archivos, y finalizando la sesión cuando la operación concluye.

La separación de responsabilidades entre el hilo que distribuye los mensajes y los hilos que ejecutan la lógica por cliente, permite escalar el sistema y asegurar una atención eficiente de múltiples clientes en simultáneo, sin bloquear el flujo general de mensajes.

3.2. Three Way Handshake

La comunicación comienza con un "three-way handshake", fundamental para asegurar una conexión confiable entre el cliente y el servidor, que 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 (`START_SESSION`):** El cliente envía un mensaje al servidor con el flag `START_SESSION`. Esto indica que el cliente desea establecer una conexión y está solicitando que el servidor también lo haga. Este mensaje incluye el comando que desea ejecutar (`upload` o `download`) y el nombre del protocolo confiable que eligió (por ejemplo, *Stop-and-Wait* o *Selective Repeat*).

2. **Aceptación de conexión (`START_SESSION_ACK`):** Si el servidor está dispuesto a aceptar la conexión, responde al cliente con un mensaje en el que establece el flag `START_SESSION_ACK`. Esto indica que el servidor está listo para recibir datos y que ha recibido correctamente la solicitud de conexión del cliente.

3. **Confirmación de conexión (`ACK`):** Finalmente, el cliente responde con un mensaje de confirmación que contiene la bandera `ACK`, indicando que recibió correctamente la respuesta del servidor y que está listo para comenzar la operación solicitada.

3.3. Manejo de Mensajes

Dado que trabajamos sobre el protocolo UDP, fue necesario construir una capa adicional que encapsule toda la información necesaria para establecer la conexión, controlar el flujo de datos y garantizar el éxito de las transferencias. Decidimos utilizar mensajes de tamaño fijo, con una estructura interna bien definida compuesta por un header de tamaño constante (414 bytes) seguido de una sección de datos de archivo de tamaño variable, con

un máximo total de 4096 bytes por mensaje. Esta decisión facilita el parsing del mensaje en ambos extremos, ya que el header puede procesarse siempre del mismo modo, sin ambigüedades. El header contiene los siguientes campos:

- `Command` (1 byte): indica la acción a realizar. `Flags` (1 byte): define el tipo de mensaje. Éstos son `START_SESSION` (mensaje inicial del cliente para solicitar conexión), `START_SESSION_ACK` (respuesta del servidor aceptando la conexión), `ACK` (confirmación del cliente o servidor de que un mensaje fue recibido correctamente), `CLOSE` y `CLOSE_ACK` (indican que una de las partes desea cerrar la conexión y confirman ese cierre), `ERROR` (para enviar mensajes de error), `LIST` (para solicitar o indicar la respuesta a un listado de archivos) y `NO_FLAGS` (para mensajes sin ninguna flag especial).
- `Data length` (4 bytes): longitud en bytes de `data`.
- `File name` (400 bytes): nombre del archivo a transferir, codificado en UTF-8 y rellenado con ceros si es más corto.
- `Sequence number` (4 bytes): número de secuencia del paquete, usado por los protocolos confiables.
- `ACK number` (4 bytes): número de acknowledgment que indica hasta qué punto se han recibido los datos.

Los 3682 bytes restantes (4096 - 414) están reservados para los datos del archivo que se desea subir o descargar. Si los datos no ocupan todo el espacio, se rellenan con ceros para mantener el tamaño fijo del mensaje. Toda la lógica de construcción y análisis de mensajes está centralizada en la clase *Message*. Ésta contiene los métodos para construir los mensajes estándar y contiene, además, dos métodos principales:

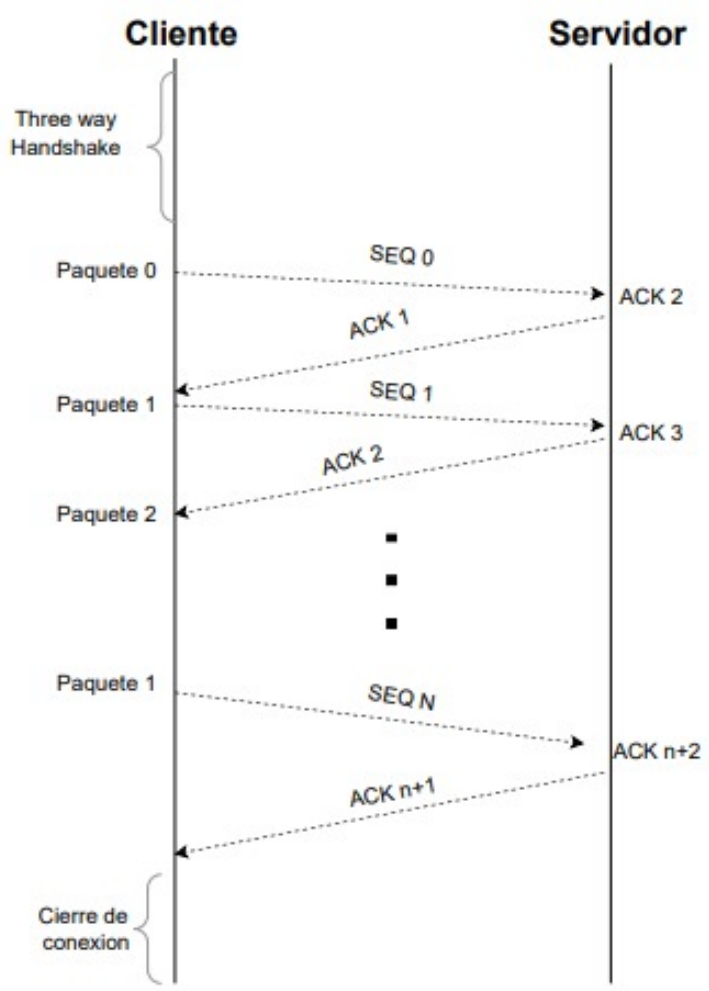
- *encode():* Serializa un objeto *Message* en un arreglo de bytes, listo para ser enviado por el socket.
- *decode():* Interpreta un arreglo de bytes recibido y lo transforma nuevamente en un objeto *Message*, extrayendo todos sus campos.

Además, esta estructura de mensajes permite integrar el protocolo confiable por encima de UDP sin depender del orden o éxito en la entrega de paquetes, ya que cada mensaje contiene suficiente información para que la otra parte pueda validar, reordenar o reenviar los datos necesarios. Finalmente, para gestionar estos mensajes durante la ejecución, el servidor utiliza colas de mensajes por cliente y crea hilos dedicados a manejar los mensajes de cada uno, asegurando que las operaciones de transferencia se realicen de forma concurrente, sin interferencia entre clientes.

3.4. Stop-and-Wait

El protocolo consiste en ir enviando de a un mensaje y no enviar el siguiente hasta no haber recibido la confirmación de que el mensaje correspondiente fue recibido (`ACK`).

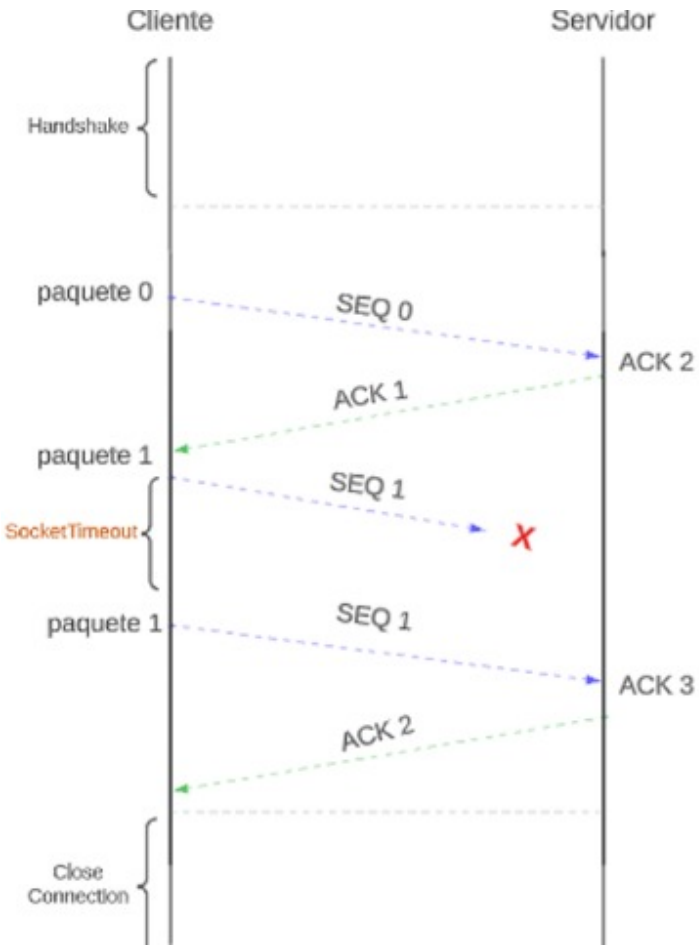
El mensaje que se envía por parte del servidor es el correspondiente al siguiente mensaje que espera recibir. Si se envía un mensaje de `ACK` con un `ACK_NUMBER` 2, significa que el servidor recibió los paquetes 0 y 1 y el siguiente que espera recibir es el paquete 2.



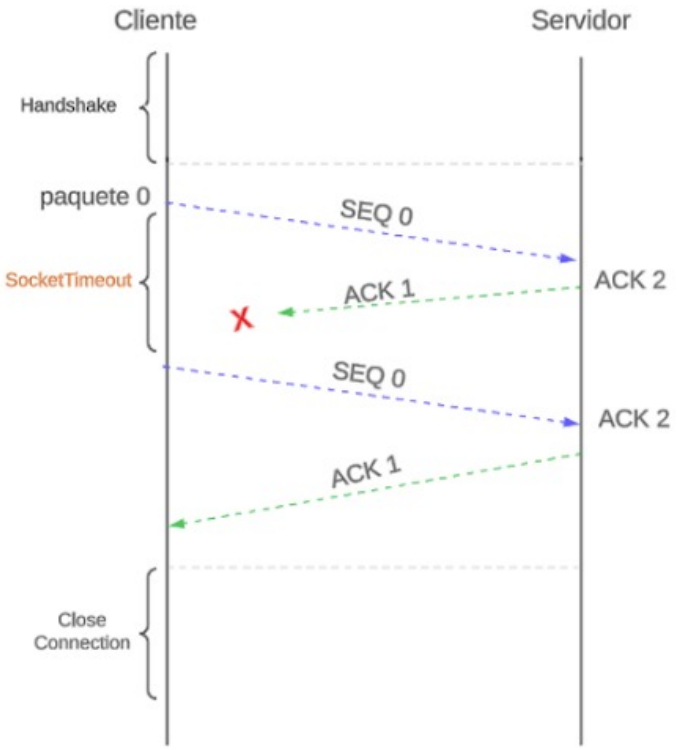
Consideraciones especiales en Stop-and-Wait

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

1. Luego de realizar el inicio de la comunicación, se envía el primer paquete. Si ese mensaje se pierde antes de llegar a destino, la parte que lo envía se queda esperando la respuesta, la cual nunca llegará. Por lo tanto, se define un *timeout*; una vez cumplido, se reenvía el mensaje, con un límite máximo de 10 reintentos hasta recibir la respuesta esperada.



2. En este escenario, el cliente ya envió el paquete y el servidor lo recibió. Una vez que lo recibe, le envía al cliente un mensaje `ACK` notificando la recepción. Si este mensaje se pierde, desde el punto de vista del cliente "se le terminó la ventana", lo cual puede indicar que su mensaje no fue recibido. Entonces, el cliente vuelve a enviar el mensaje anterior. Como el servidor ya lo había recibido, le responde que ese mensaje ya fue recibido y que espera el siguiente. Si el mensaje del servidor llega correctamente, el cliente continúa con el siguiente envío. Al igual que en el caso anterior, existen 10 instancias de reintento para cada mensaje.



3.5. Selective - Repeat

En este protocolo se utilizan dos ventanas, una del lado del emisor y otra del lado del receptor. El emisor puede enviar múltiples paquetes sin esperar a que todos sean confirmados, y el receptor puede aceptarlos incluso si llegan fuera de orden. Esto nos permite aprovechar mejor el canal de comunicación y mejorar el rendimiento, sobre todo cuando existe latencia o pérdida de paquetes en la red.

La ventana del emisor cumple dos roles principales:

1. Útil para rastrear las confirmaciones de recepción (ACKs) de los paquetes enviados.
2. Limita la cantidad de paquetes en camino al restringir cuántos paquetes pueden ser enviados antes de recibir sus respectivos ACKs.

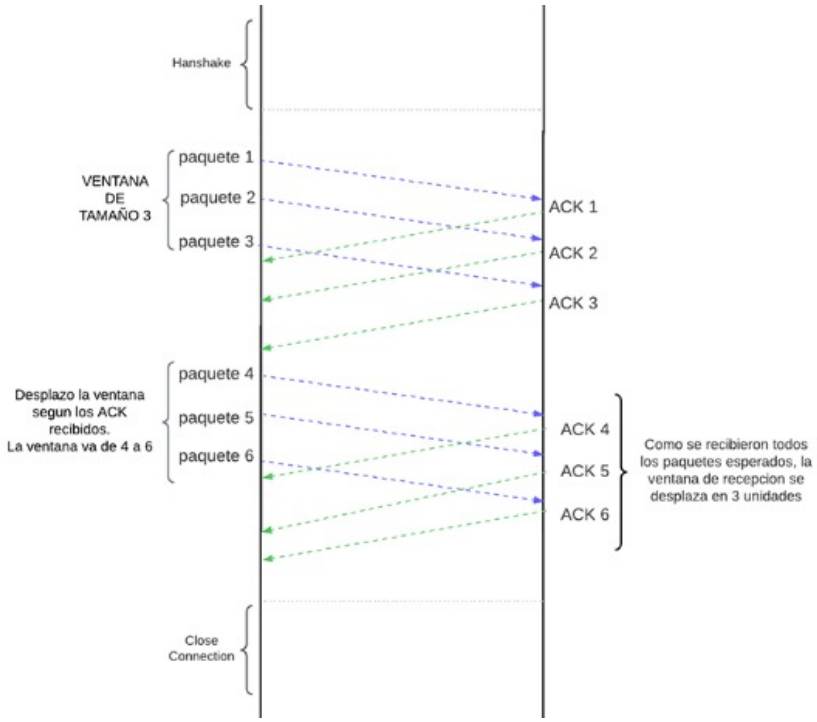
Cuando el emisor envía mensajes, estos deben estar dentro de su ventana. A medida que recibe los ACKs de dichos mensajes, los marca como recibidos.

Esto permite enviar más mensajes. Si el ACK que llega es del primer paquete de la ventana, se mueve la ventana, lo cual garantiza que no se sature la red con muchos paquetes no confirmados (ACKs).

La ventana del receptor se encarga de llevar un registro de los paquetes esperados. Si un mensaje llega fuera del rango de su ventana, el receptor lo descarta, pero 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 que el paquete no sea el esperado según el orden.

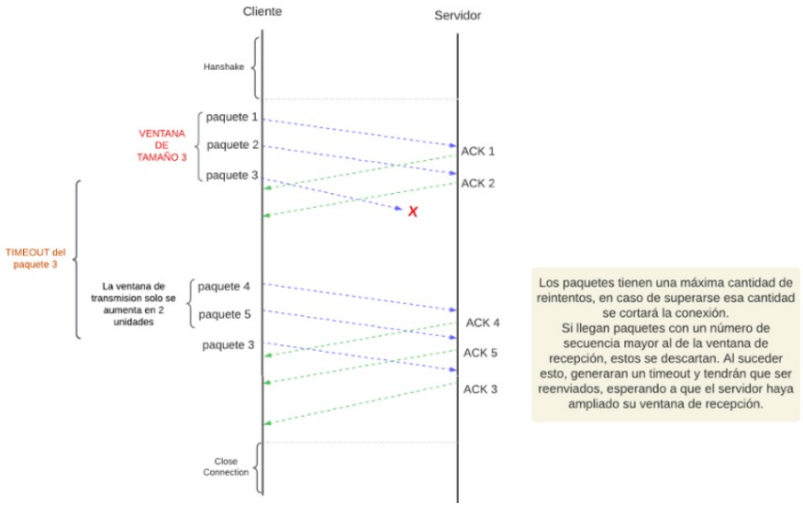
A medida que llegan los mensajes que el receptor espera, amplía su ventana para acomodarlos. Cuando finalmente recibe un mensaje en el orden esperado, procesa el buffer para asegurar que los datos se procesen en el orden correcto, garantizando así la integridad y el orden de la comunicación.

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.



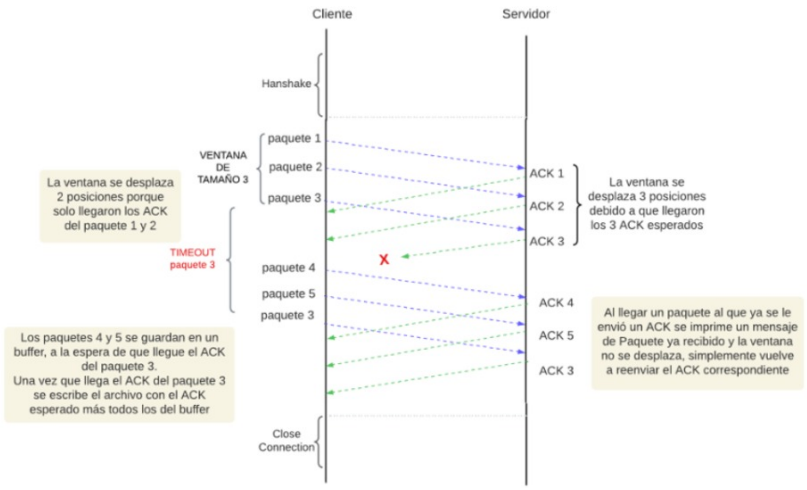
Cuando se pierde un paquete, detectado gracias a su propio timeout, se envía solamente el que se perdió.

Los paquetes tienen una máxima cantidad de reintentos; si se supera, se corta la conexión. Si llegan paquetes con un número de secuencia mayor al de la ventana de recepción, se descartan, lo que generará un timeout y tendrán que ser reenviados, a la espera de que el servidor haya ampliado su ventana de recepción.



En el siguiente escenario, el paquete 3 es recibido pero el ACK enviado por el servidor no es recibido por el cliente, lo que genera que la ventana del cliente se desplace en 2 posiciones enviando los paquetes 4 y 5, debido a que recibió los ACK del 1 y 2.

Una vez que se genera el timeout para el paquete 3 se reenvía el paquete. Como los ACK de los paquetes 4 y 5 fueron recibidos primero que el del paquete 3, se almacenan en el buffer, a la espera de que llegue el ACK del paquete 3. Una vez que llega el ACK del 3 se persisten el recibido y todos los que estaban en el buffer, es decir 3, 4 y 5.



3.6. Cierre de Conexión

Dado que estamos trabajando sobre UDP —que no tiene ningún mecanismo de cierre de conexión incorporado como TCP— debimos crear un protocolo de finalización con los flags `CLOSE` y `CLOSE_ACK`.

En nuestro trabajo, el encargado de iniciar el cierre es quien envía el archivo. Entonces:

- En una operación de `UPLOAD`, el cierre es iniciado por el cliente.
- En una operación de `DOWNLOAD`, lo inicia el servidor.

Una vez enviados todos los paquetes y recibidos sus respectivos `ACKs`, el emisor envía un mensaje con la flag `CLOSE` mediante el método `send_close_and_wait_for_response`. La parte receptora, al recibir este mensaje, interpreta que no habrá más datos restantes y responde con un mensaje `CLOSE_ACK`.

El emisor se queda esperando el `CLOSE_ACK` con un timeout y un máximo de 10 reintentos (definidos con `MAX_TIMEOUT_RETRIES`). Si no recibe la confirmación en ese tiempo, vuelve a enviar el `CLOSE`, repitiendo este ciclo hasta recibir una respuesta válida o alcanzar el número máximo de intentos.

Una vez completado este intercambio, cada extremo cierra su socket y libera cualquier recurso asociado a la transferencia. Aplicamos este procedimiento tanto para `Selective Repeat` como para `Stop-and-Wait`, unificando el comportamiento del sistema y asegurando un cierre robusto en cualquier situación.

4. Pruebas

No basta con implementar el protocolo: es fundamental validar su comportamiento tanto en entornos ideales como en contextos con condiciones reales de red. Para ello, realizamos pruebas tanto en entorno `localhost` como en un entorno simulado de red utilizando `Mininet`, donde se puede configurar una tasa de pérdida de paquetes específica.

Estas pruebas permiten:

- Comprobar la fiabilidad de la transferencia de archivos cuando hay pérdida de paquetes.
- Comparar la eficiencia entre `Stop and Wait` y `Selective Repeat`.

- Medir el tiempo de transferencia para archivos de distintos tamaños y bajo diferentes porcentajes de pérdida de paquetes.

Entorno de pruebas con Mininet

Para simular condiciones más cercanas a una red real, se utilizó la herramienta `Mininet`. Esta plataforma permite levantar el servidor y el cliente en nodos distintos dentro de una red virtual configurable. Allí se puede establecer, por ejemplo, un 5 %, 10 % o 20 % de pérdida de paquetes, además de latencias fijas. Esto resulta útil para observar diferencias claras entre los protocolos implementados.

4.1. Mediciones

Cuadro 1: Tiempos de transferencia para diferentes protocolos y tamaños de archivo

Protocolo	**Dirección**	**5 MB**	**1 MB**
2* Stop and Wait	Upload	2,66 s	0,47 s
	Download	7,55 s	5,47 s
	Upload (10 % loss)	64,83 s	12,08 s
	Download (10 % loss)	35,75 s	11,76 s
3* Selective Repeat	Upload	0,48 s	0,04 s
	Download	0,51 s	0,13 s
	Upload (10 % loss)	21,03 s	4,14 s
	Download (10 % loss)	21,89 s	3,87 s

5. Análisis

El análisis confirma que sin pérdida ambos protocolos rinden similar. Con latencia, Selective Repeat es más rápido; con pérdida, Selective Repeat supera ampliamente a Stop-and-Wait.

6. Preguntas

6.1. Describa la arquitectura Cliente-Servidor

La arquitectura cliente-servidor es un modelo comúnmente utilizado en sistemas distribuidos, donde se separan claramente dos entidades, cada una con sus propias funciones y responsabilidades. Los clientes y servidores se comunican a través de la red.

Por un lado, el servidor es el encargado de responder las solicitudes de los clientes con los recursos y servicios que este provee. Se lo diseña de forma tal que sea capaz de atender las consultas de múltiples clientes a la vez. El servidor aguarda, de manera pasiva, por solicitudes de los clientes.

Por otro lado, el cliente es aquel componente que inicia la comunicación entre las partes solicitando servicios o recursos al servidor. Suele ser una aplicación ejecutada por el usuario final. El cliente inicia un request o pedido, el servidor la recibe, procesa y finalmente le devuelve una respuesta al cliente quien la utiliza según necesite.

Las características más importantes de esta arquitectura giran en torno al desacoplamiento, que puede ser tanto físico como lógico de las entidades, y a la centralización de los recursos del servidor, para que estos puedan ser administrados de manera eficiente y segura.

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

La función principal de un protocolo de capa de aplicación es permitir la comunicación entre aplicaciones que se ejecutan en dispositivos distintos a través de una red, definiendo reglas claras que deben seguirse para que el intercambio de datos sea correcto.

Algunas reglas a definir en un protocolo de capa de aplicación se centran en cómo las aplicaciones intercambian datos y de qué forma deben ser los mensajes, para que cada entidad los pueda entender y procesar la información recibida. Se encargan de estandarizar la comunicación.

Existen varios protocolos de capa de aplicación; entre los más utilizados hoy en día se encuentran HTTP (Hypertext Transfer Protocol) o FTP (File Transfer Protocol), por ejemplo. Cada uno es específico para distintos tipos de aplicación, por lo que la sintaxis solicitada y el formato de las peticiones y respuestas varía en cada caso.

6.3. Detalle el protocolo de aplicación realizado en este trabajo

Se desarrolló un protocolo de aplicación orientado al protocolo FTP (File Transfer Protocol), siendo la transferencia de archivos entre diferentes aplicaciones (cliente y servidor) el foco principal de este desarrollo.

Como se explicó en la sección 3, implementamos un sistema que incluye un *three-way handshake* para establecer la conexión, mensajes con estructura fija para intercambiar datos y dos protocolos confiables a elección del cliente: `Stop-and-Wait` y `Selective Repeat`. Además, el cierre de la sesión se realiza mediante un intercambio de mensajes `CLOSE` y `CLOSE_ACK`.

6.4. 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 del modelo TCP/IP se encarga de garantizar que los datos enviados desde una aplicación en un dispositivo lleguen correctamente a otra aplicación en otro dispositivo. Para esto, ofrece dos protocolos principales: TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). Ambos cumplen esta función, pero lo hacen de formas diferentes y con características distintas.

TCP es un protocolo orientado a la conexión, lo que significa que antes de comenzar la transmisión de datos, debe establecerse una conexión entre el emisor y el receptor para asegurar que estos se puedan comunicar correctamente. Este proceso se conoce como *three-way handshake*. Primero, el cliente envía un mensaje para alertar al servidor que se quiere conectar; el servidor le envía un mensaje avisando que reconoció su solicitud de conexión, y por último el cliente le confirma que recibió este mensaje del servidor.

Una vez establecida la conexión, TCP proporciona un canal confiable y ordenado, ya que garantiza que los datos lleguen completos, sin errores y en el mismo orden en

que fueron enviados. Para lograr esto, el protocolo implementa mecanismos de control de errores, como el uso de números de secuencia y acuses de recibo (ACK), y puede retransmitir paquetes perdidos.

TCP cuenta también con mecanismos de control de flujo para evitar que el emisor sature al receptor, y control de congestión para reducir la sobrecarga en la red en situaciones de tráfico elevado. Debido a estas características, TCP es ideal para aplicaciones donde la integridad de los datos y el orden son esenciales, como la navegación web (HTTP/HTTPS), la transferencia de archivos (FTP), el correo electrónico (SMTP), entre otras aplicaciones que requieren alta fiabilidad.

UDP, en cambio, es un protocolo no orientado a la conexión. No establece una comunicación previa entre emisor y receptor antes de enviar los datos. Transmite directamente los paquetes (datagramas), pero no verifica si llegaron correctamente, en qué orden lo hicieron, o si hubo pérdidas. UDP tampoco implementa mecanismos de control de flujo ni de congestión, por lo que el emisor puede enviar datos al ritmo que desee, sin importar si el receptor o la red son capaces de manejarlos o no.

Esta simplicidad reduce significativamente la sobrecarga del protocolo, lo que lo hace más rápido y eficiente en términos de uso de recursos. Sin embargo, al no garantizar la entrega ni el orden de los paquetes, UDP es considerado un protocolo no confiable.

A pesar de ello, resulta especialmente útil en contextos donde la velocidad es prioritaria y la pérdida ocasional de datos no afecta gravemente la experiencia del usuario. Ejemplos típicos de este tipo de aplicaciones son la transmisión de audio y video en tiempo real, las videollamadas (VoIP), los videojuegos en línea y los servicios de DNS, donde los tiempos de respuesta rápidos son fundamentales.

Resumen:

- **TCP:** orientado a la conexión, confiable, ordenado, con control de flujo y congestión. Usado en aplicaciones donde la precisión y el orden son claves.
- **UDP:** no orientado a la conexión, rápido pero no confiable ni ordenado. Usado en aplicaciones donde la velocidad es prioritaria y se toleran pérdidas.

7. Dificultades

Detallamos a continuación algunas dificultades a las que nos enfrentamos durante el desarrollo de este trabajo.

Por un lado, la implementación del protocolo `Selective Repeat` significó un desarrollo mucho más complejo y extenso que el requerido para `Stop and Wait`. Nos encontramos lidiando con varias problemáticas al tener que utilizar múltiples hilos concurrentemente para la transferencia de varios paquetes en simultáneo y manejar las confirmaciones (`ACKs`) de los paquetes enviados; lo que nos llevó a tener que analizar en profundidad cómo manejar las problemáticas asociadas a condiciones de carrera (*race conditions*).

Una dificultad crítica se presentó en el método `join_and_cleanup_ack_thread`, responsable de limpiar correctamente los hilos que ya hayan cumplido su función. El problema en cuestión era que, en algunas ocasiones, el `ACK` correspondiente llegaba, pero debido a una condición de carrera entre la recepción del `ACK` y la inicialización completa del hilo asociado (el mapeo de `acks_map[ack_number]`), este aún no existía en el momento en el que se intentaba acceder. Esto derivaba en errores en los que, en ciertas ejecuciones, el sistema funcionaba correctamente, y en otras fallaba sin una causa aparente, ya que la lógica era válida pero los tiempos entre hilos no se cumplían según lo esperado.

Este problema se resolvió mediante la implementación de una lógica de reintento con *backoff* suave: si en la primera consulta no se encontraba el hilo o su cola asociada, el sistema esperaba un pequeño intervalo de tiempo y volvía a intentar, hasta alcanzar un número máximo de reintentos. Esto permitió compensar los pequeños desfases temporales entre el envío del paquete, la creación del hilo de espera y la llegada del `ACK` correspondiente. Todo esto se logró manteniendo la lógica no bloqueante del sistema, eliminando fallas no deterministas y evitando la pérdida de paquetes por errores de sincronización interna.

Por otro lado, la instalación y uso de la herramienta `Mininet` no fue inicialmente sencilla de comprender. Se presentaron diversas complejidades y problemáticas, dado que dicha herramienta no es compatible con las versiones más recientes de `Python`. La gestión de entornos en la máquina virtual resultó poco intuitiva y dificultó el proceso de adaptación del sistema a los requerimientos del software. A esto se sumó que `Mininet` solo podía ejecutarse dentro de dicha máquina virtual, lo que agregaba una capa adicional de complejidad y volvía el acceso a la herramienta más lento y menos eficiente.

Finalmente, en nuestra experiencia personal, comprender a fondo la tipología del lenguaje `Python` también representó un desafío, ya que ninguno de los integrantes había realizado proyectos extensos con este lenguaje en otras materias de la facultad ni en proyectos personales previos.

8. Conclusión

En este trabajo práctico desarrollamos una aplicación para la transferencia confiable de archivos utilizando el protocolo de transporte `UDP`, sobre el cual construimos nuestra propia capa de control de errores. Implementamos dos versiones del protocolo: `Stop-and-Wait` y `Selective Repeat`.

A lo largo del desarrollo nos encontramos con distintos desafíos, como el diseño del formato de mensajes, el manejo de hilos concurrentes, la sincronización entre envíos y confirmaciones, y la gestión de cierres de conexión robustos en un entorno sin garantías de entrega como `UDP`. La implementación de `Selective Repeat`, en particular, requirió especial atención para evitar condiciones de carrera y pérdidas de paquetes por problemas internos de sincronización.

Además de construir un sistema funcional, realizamos pruebas sobre entornos simulados con `Mininet` para analizar el comportamiento de ambos protocolos bajo distintos porcentajes de pérdida de paquetes. La instalación y configuración inicial de `Mininet` resultó ser un desafío en sí mismo, ya que la herramienta no era compatible con las versiones actuales de `Python` ni con nuestro entorno local. Esto nos obligó a aprender a utilizar máquinas virtuales específicas, gestionar entornos aislados y entender el flujo de red dentro de un entorno simulado, lo cual requirió tiempo y una curva de aprendizaje considerable. De todas formas, `Mininet` nos permitió testear el sistema en condiciones realistas de pérdida y retraso.

Este proyecto nos permitió profundizar en el funcionamiento interno de los protocolos de capa de aplicación, comprender cómo construir una comunicación confiable sobre `UDP` y ganar experiencia en el diseño de sistemas distribuidos concurrentes y tolerantes a fallas. Sin duda, el trabajo nos desafió en múltiples niveles técnicos, desde la teoría de redes hasta la programación multihilo en `Python`, y nos dejó una base sólida para futuros desarrollos en sistemas de comunicación.