Introducción a los Sistemas Distribuidos
(75.43) Curso Hamelin

Trabajo Práctico 1
File Transfer

8 de mayo de 2025

Leticia Figueroa
110510

Andrea Figueroa
110450

Josue Martel
110696

Jesabel Pugliese
110860

Kevin Vallejo
109975

Índice

1. Introducción

El presente documento parte de la entrega del primer trabajo grupal de la materia Introducción a los Sistemas Distribuidos (75.43/75.33/95.60). Los aspectos a tratar se subdividen en dos secciones:

File Transfer Over UDP (FTOU)

Se pidió desarrollar una aplicación de arquitectura cliente-servidor que implemente la funcionalidad de transferencia de archivos mediante las siguientes operaciones:

- **Upload:** transferencia de un archivo del cliente hacia el servidor
- **Download:** transferencia de un archivo del servidor hacia el cliente

Se pidió el diseño e implementación de un protocolo de aplicación básico que especifique los mensajes intercambiados entre los distintos procesos. La comunicación entre los procesos se debió implementar utilizando UDP como protocolo de capa de transporte. Para lograr una transferencia confiable al utilizar el protocolo UDP, se pidió implementar una versión utilizando el protocolo *Stop & Wait* y otra versión utilizando el mecanismo de error-recovery de *Selective Repeat*.

Anexo: proceso de fragmentación de IPv4

Se pidió armar una red virtual para la puesta en práctica de los conceptos y herramientas necesarias para la comprobación del proceso de fragmentación en IPv4.

2. Hipótesis y suposiciones realizadas

Hipótesis

Las hipótesis planteadas que trataremos de demostrar empíricamente son:

- FTOU es un sistema robusto, capaz de soportar fallas en la conexión con un cliente sin dejar de atender a los demás clientes con los que mantenga una conexión.
- FTOU garantiza que un cliente y un servidor puedan conectarse a través de un canal de comunicación lógico único para este flujo, es decir, que un cliente A conectado con el servidor no interfiere en la comunicación de otro cliente con su propia conexión al servidor.
- FTOU garantiza que, dado un cliente conectado directamente con el servidor, para archivos de 5MB, la transferencia se completa en 2 minutos o menos.
- FTOU contempla condiciones de error tales como pérdida de paquetes, envíos desordenados de paquetes, paquetes corruptos y delay.

Suposiciones

Para un correcto funcionamiento, nuestro programa realiza las siguientes suposiciones:

- Los sistemas terminales cuentan con la capacidad espacial para almacenar los archivos que deseen recibir. Esto abarca el espacio de almacenamiento que dispone el servidor para los archivos recibidos por los clientes *upload* y el espacio que dispone un cliente download para almacenar los archivos recibidos por el servidor.

- La cantidad de clientes en comunicación con el servidor nunca supera la cantidad de puertos disponibles.
- Tanto cliente como server terminan una comunicación establecida de forma *protocolar*. Es decir, ni servidor ni cliente terminan el programa matando el proceso.
- Durante la ejecución del programa servidor, ningún otro programa externo a los presentados tratará de atacar el servidor mandando mensajes a puertos al azar del servidor en búsqueda de interferir con el flujo de comunicación establecido.
- Multiples clientes en el mismo host no intentarán descargar el mismo archivo a la misma carpeta con el mismo nombre de archivo.
- Los mesajes que el servidor recibe al puerto público son únicamente clientes que quieren estabecer una conexión, es decir, siguen el protocolo establecido para poder realizar uso de los servicios provistos.
- La red sobre la cual se transfieren los datos no es sensible a ser saturada por lo que no es necesario ajustar dinámicamente la tasa de envío (no es necesario fast recovery ni quick start porque la cantidad de usuarios contectados a la red no es inconveniente)
- El largo de cualquier archivo a transferirse nunca sobrepasa los 268431360 bytes

3. Implementación

En la presente sección se expondrá la estructura clave del sistema cliente-servidor de transferencia de archivos implementado. Este mismo consta de tres aplicaciones, dos de cliente y una de servidor.

Aplicación	**Punto de entrada**
Servidor	`/src/start-server.py`
Cliente Download	`/src/download.py`
Cliente Upload	`/src/upload.py`

Cuadro 1: Puntos de entrada de las aplicaciones

Antes de explicar detalles de la implementación presentada, procederemos a exponer, los roles en la comunicación a nivel procesos, los flujos de comunicación principales, así como las estructuras empleadas para poder llevar a cabo la comunicación.

3.1. Roles de comunicación entre procesos

3.2. Flujo de comunicación Cliente-Servidor

La comunicación entre cliente y servidor se subdivide en dos partes. La diferencia práctica entre estas dos partes se concretiza en el uso de una flag (SYN) ubicado en el primer byte de todos los mensajes del flujo de comunicación (sea apagada o encendida dependiendo del caso). Esta única flag nos permite hacer la siguiente distinción;

Comunicación	**Sender**	**Receiver**
Cliente Download – Servidor	Servidor	Cliente Download
Cliente Upload – Servidor	Cliente Upload	Servidor

Cuadro 2: Roles de comunicación entre procesos

- **Handshake**: Esta etapa inicial hace alusión al *3-way handshake* de TCP, implica el intercambio de mensajes independientes del protocolo de recuperación de errores. Son los únicos en todo el flujo de comunicación que incluyen la flag *SYN* activada, y transportan la metadata necesaria para establecer correctamente la conexión entre los hosts.

- **Flujo de datos**: Una vez completado el handshake, los mensajes se ajustan al esquema del protocolo de recuperación. A partir de este punto, todos los segmentos tienen la bandera *SYN* desactivada. Los mensajes enviados por el *proceso sender* (ya sea el servidor o el cliente) contienen datos de la capa de aplicación, es decir, *chuncks* del archivo en transferencia.

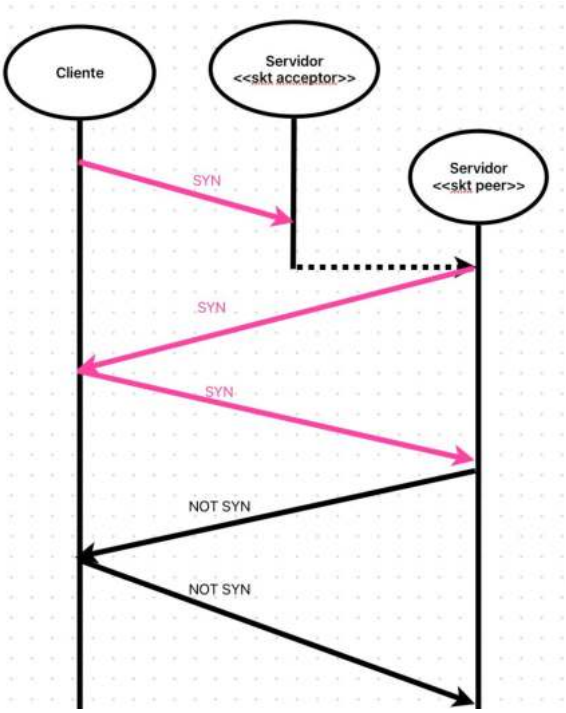


Figura 1: Flujo general de la comunicación

Particularmente, este flujo general se puede especializar para ser más preciso en base a los los roles de comunicación. Veamos que el *sender* siempre envia el primer mensaje *NOT SYN*

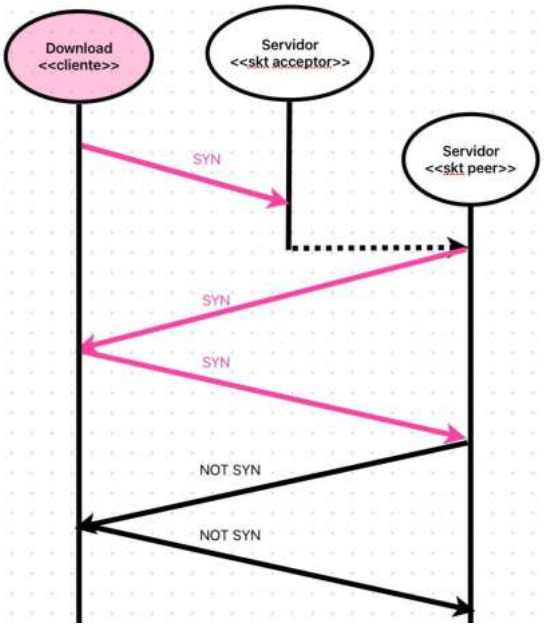


Figura 2: Flujo de comunicación con un Cliente Download

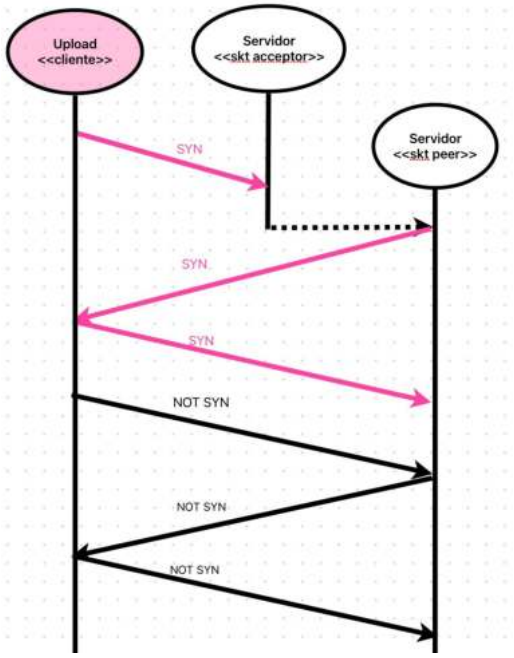


Figura 3: Flujo de comunicación con un Cliente Upload

3.3. Arquitectura de software

La aplicación Servidor proporciona el servicio de almacenamiento y transferencia segura de archivos binarios a aquellos otros dispositivos que requieran del servicio (clientes). Es decir, opera pasivamente frente a las peticiones que le lleguen de los clientes. Bajo la arquitectura implementada

(cliente-servidor), nuestro diseño busca que el servidor sea capaz de ofrecer a múltiples clientes la posibilidad de ser atendidos, conectarse con el servidor y realizar una operación Upload/Download. Para lograr nuestro cometido sin tiempos de espera secuenciales, la organización de los flujos de **ejecución** que maneja el servidor es la siguiente:

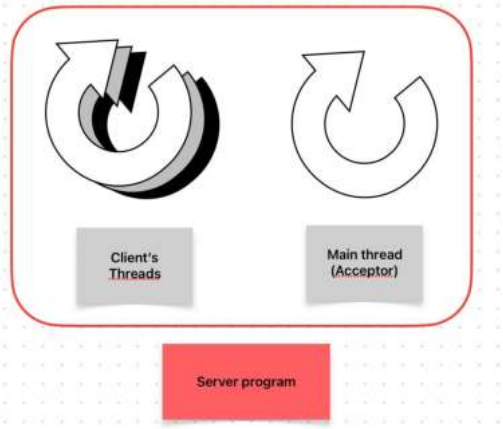


Figura 4: Flujo de comunicación con un Cliente Upload

3.3.1. Reserva de recursos

El proceso previo al handshake que sigue el servidor para reservar los recursos dedicados a cada conexión, se desgloza en los siguientes dos pasos:

1. **Escucha solicitudes de conexión:** En el hilo de ejecución principal del proceso se abre un puerto de red (asociado al protocolo de transporte UDP) que permitirá a este mismo recibir solicitudes de *conexión* (primer mensaje *SYN* con el formato antes referido: Fig. 5). A este puerto lo denominamos *socket UDP listener* o simplemente *socket acceptor*.

2. **Aceptar clientes:** Posteriormente el servidor entra en un bucle que en cada iteración obtiene un mensaje de un client por este puerto: Por cada nuevo mensaje que se obtenga de dicho puerto "público", si es un cliente nuevo, se crea un *Thread* que dispondrá de su propio *socket UDP* (denominado *peer*) dedicado a continuar el flujo de comunicación. El *socket peer* de ahora en más será el socket con el que se conectará el cliente, de forma que el socket acceptor no se ocupa de otros mensajes que no sea de *solicitud de conexión*.

3.3.2. Limpieza de recursos

Debido a que por cada cliente nuestro servidor reserva recursos para poder atenderlo, la forma en cómo realizamos la limpieza de los recursos asociados a un cliente con el cual la conexión ya terminó se realiza a través de un efectivo *ReapDead*. Basicamente, en cada iteración verificamos a los clientes que ya terminaron su comunicación y por cada uno de estos liberamos el thread y el puerto que empleaba este.

3.4. Esquema general de mensajes *SYN*

De aquí en adelante, al referir a algún mensaje referente al handshake, hacemos referencia a las siguientes secuencias de bytes y su respectiva interpretación:

Header	**Longitud**	**Rango de valores**	**Intención**
SYN	1 byte	1 o 0	Indica si el mensaje es o no es un mensaje del Handshake
Número de secuencia	1 byte	0	Indica el número secuencial con el que el programa inicia el proceso de comunicación.
Tipo de protocolo	2 bytes	'sr' o 'sw'	Indica el tipo de protocolo de recuperación que emplea el cliente y con el cual se espera que el servidor responda.
ACK	1 byte	0	Indica el acknowledge del número de secuencia recibido del otro punto de comunicación en el mensaje previo.

3.4.1. Primer mensaje *SYN*

El Cliente contacta al servidor, para ello envía al menos 4 bytes (además de la metadata de la capa de aplicación).

- Enviado por: Programa cliente (Download o Upload)

- Recibido por: Programa servidor (Main thread por socket acceptor)

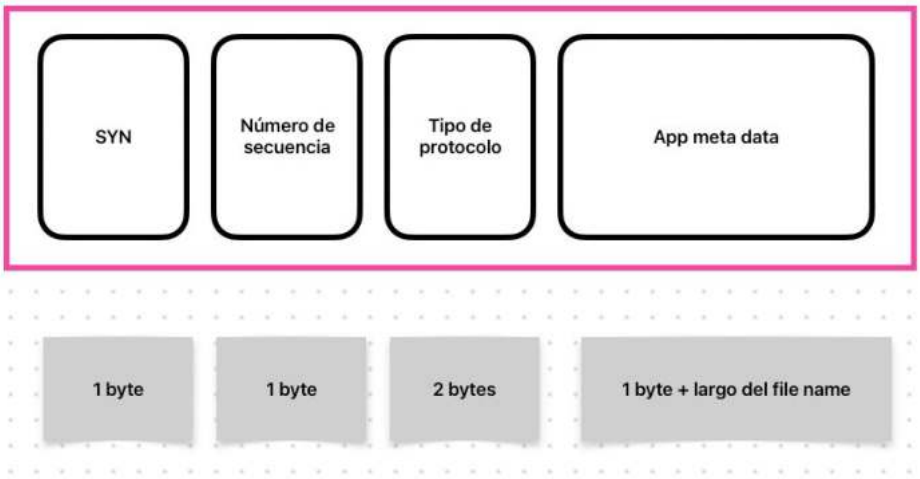


Figura 5: Primer mensaje SYN

3.4.2. Segundo mensaje *SYN*

El Servidor responde al primer mensaje *SYN* desde el hilo dedicado al cliente en particular, esto implica que el socket empleado para enviar este mensaje no es el socekt acceptor, es el socket peer de este flujo.

- Enviado por: Programa Servidor (Client's thread)

- Recibido por: Programa cliente

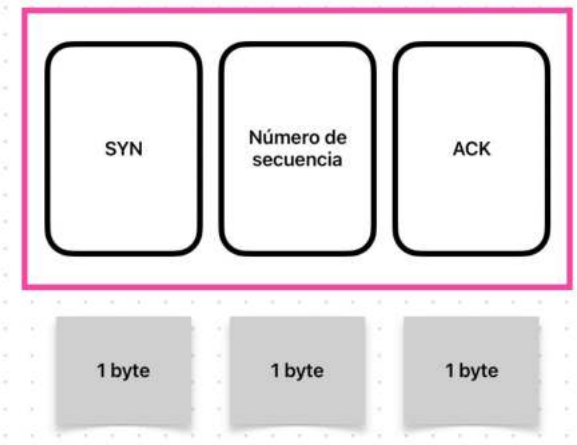


Figura 6: Segundo mensaje SYN

3.4.3. Tercer mensaje *SYN*

El Cliente confirma el recibimiento del segundo mensaje *SYN*, por lo que esa confirmación se interpreta como el establecimiento protocolar de la conexión exitosa con el servidor.

- Enviado por: Programa cliente (Download o Upload)

- Recibido por: Programa servidor (Client's thread por socket peer)

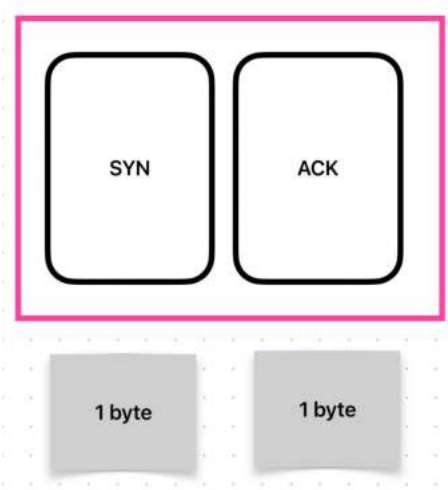


Figura 7: Tercer mensaje SYN

3.5. Handshake en acción

Si el envío de mensajes se consigue sin pérdida de paquetes, el flujo de mensajes es el mismo al ya mencionado previamente. Por otro lado, en un contexto donde cada uno de estos mensajes se puede perder, observamos que nuestro sistema robusto intenta establecer la conexión a pesar de la pérdida de algunos mensaje:

Pérdida del primer mensaje *SYN*

- **Reacción del cliente**: Intenta contactar al servidor en busqueda de ser *aceptado*, esto se repite una cantidad arbitraria de veces como máximo, si este límite se alcanza se considera que la conexión fue rechazada.
- **Reacción del server**: Si el mensaje no llega, el servidor no se dará cuenta que un cliente no se pudo contactar. Si el mensaje llega en duplicado (producto de un reenvio) entonces el servidor evita reservar recursos innecesarios, es decir si ya hay un thread que se haya derivado para la comunicación con el cliente.

Pérdida del segundo mensaje *SYN*

- **Reacción del cliente**: El cliente no obtiene respuesta por lo que recurre al puerto público para reintentar ser atendido. Después de un número arbitrario de intentos, el cliente se rinde y entiende que la conexión falló.
- **Reacción del servidor**: El servidor reintenta contactar al cliente hasta obtener una respuesta o hasta agotar la cantidad de intentos establecida. Si se excede del límite, se asume que la conexión no se pudo establecer; si recibe un mensaje, entonces el servidor asume que la conexión con el cliente se logró establecer.

Por ejemplo si el cliente es de Download (resulta idéntico para un cliente Upload)

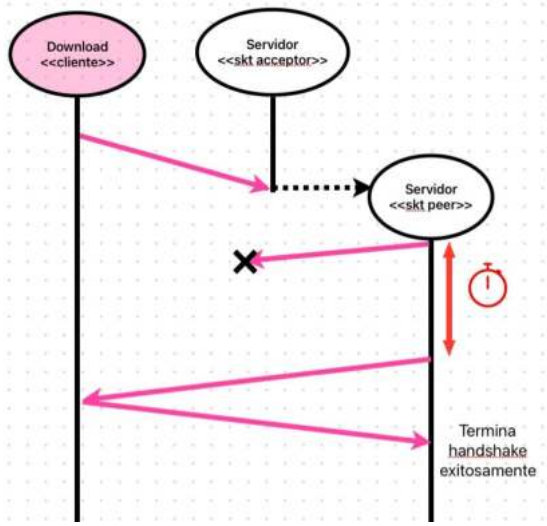


Figura 8: Conexión recuperada ante pérdida del segundo mensaje.

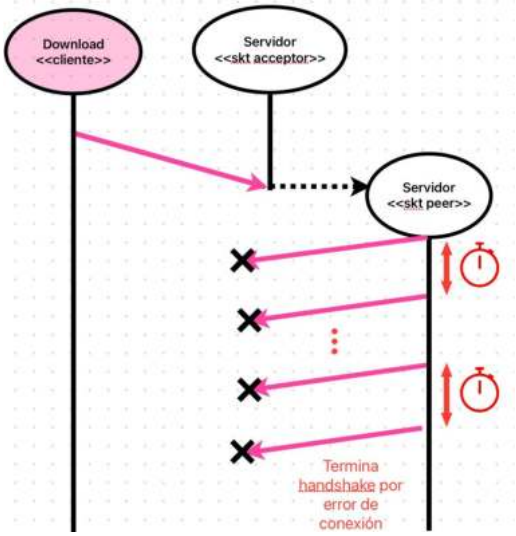


Figura 9: Error de conexión: pérdida definitiva de la comunicación.

Pérdida del tercer mensaje *SYN* - Cliente Download Si la comunicación se realiza entre el servidor y un Cliente Download, si el mensaje que el cliente envió se pierde, el cliente insistirá hasta saber que su mensaje llegó, es decir, continua enviando el tercer mensaje SYN hasta que sepa que su mensaje llego, para entra en un loop que itera cierta cantidad de intentos en los que intenta recibir algo.

- Si llega un mensaje *SYN* lo ignora (es el servidor reenviando el segundo mensaje *SYN*) y continua en sus intentos por confirmar que su mensaje llegó.
- Si llega el límite establecido de timeouts sin haberse recibo algún mensaje, asume que es un **error de conexión** y no se pudo establecer comunicación con el server.
- Si llega un mensaje*NOT SYN*, el handshake **termina exitosamente**, el servidor empezó a enviar datos.

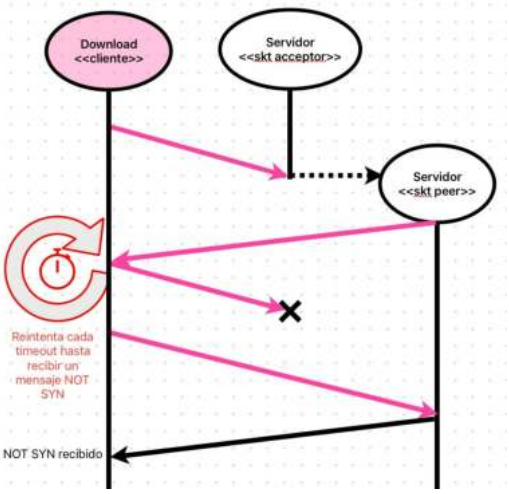


Figura 10: Conexión recuperada ante pérdida del tercer mensaje.

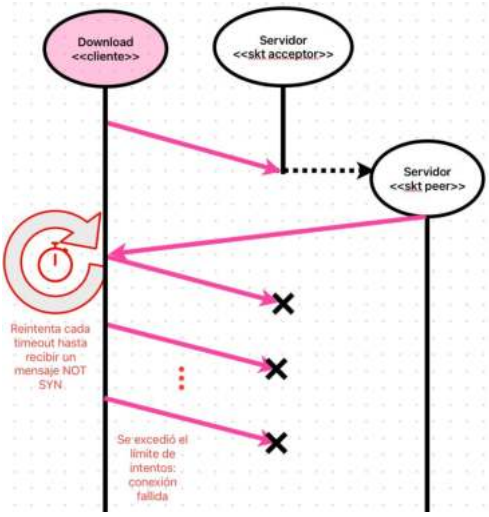


Figura 11: Error de conexión: pérdida definitiva de la comunicación.

Pérdida del tercer mensaje *SYN* - Cliente Upload Si la comunicación se realiza entre el servidor y un cliente Upload, el cliente envía el tercer mensaje *SYN* una sola vez y termina el handshake para este. Por otro lado, el servidor estará esperando o la respuesta *SYN* que se perdió o el primer mensaje *NOT SYN* que se envía a través del Reliable Data Transfer escogido. En cualquier caso, establece la conexión con éxito.

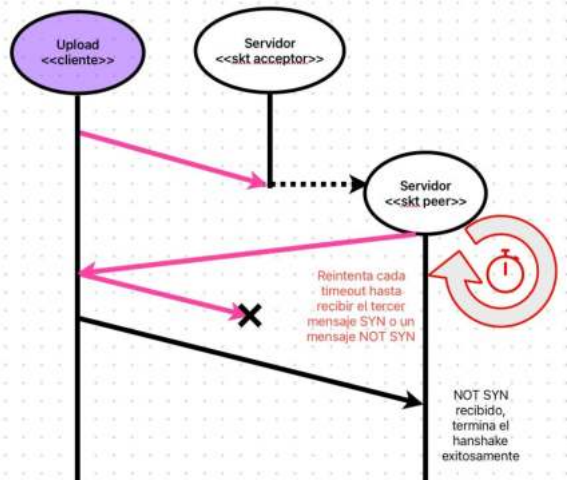


Figura 12: Conexión recuperada ante pérdida del tercer mensaje

3.6. Conexión establecida: Mensajes NOT SYN

Una vez terminado el handshake los sistemas terminales proceden con el intercambio de mensajes que permitan transferir el archivo deseado. El contendio del archivo en cuestión tiene que se transferido en su totalidad internamente cumpliendo el esquema general propuesto de los segmentos. A continuación se procede a exponer el esquema de los segmentos que se deben de enviar por la red, y posteriormente se describirá desde un approach top-down: como las aplicaciones inician el envio-recepcion del archivo en cuestión haciendo uso de sus RDTS y finalmente se detallará la lógica subyacente de los rdts: los protocolos de recuperación de errores, cómo se envían los mensajes en segmentos y cómo se aseguran que lleguen al otro extremo de la comunicación.

3.6.1. Visualización de las componentes de un mensaje

- Mensajes del sender

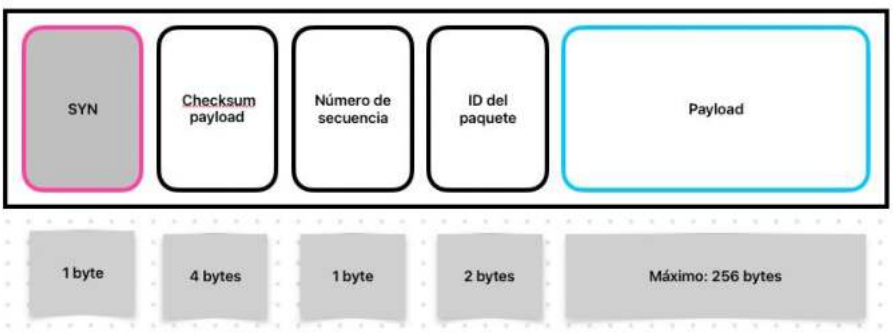


Figura 13: Mensaje del sender

- Mensajes del receiver

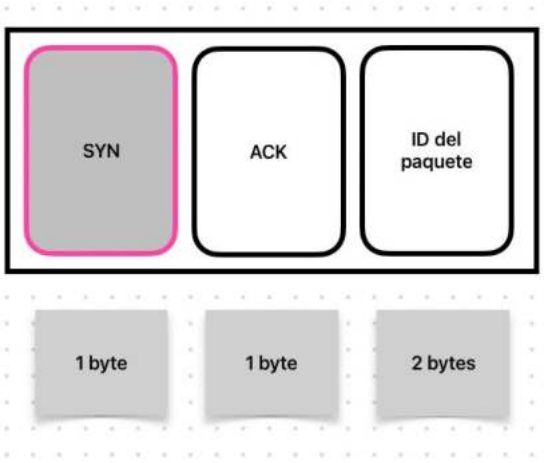


Figura 14: Acuse de conocimiento del receiver

3.7. Transferencia a nivel capa de Aplicación

Una vez confirmada la conexión se procede a inicializar el protocolo para la recuperación de errores del *Reliable Data Transfer* de cada aplicación y se procederá con la transferencia del archivo en cuestión desde-al servidor.

Tanto Server como Cliente desarrollan la comunicación indicando el address que el handshake indica como la address destinada al flujo. El objetivo En la capa de Aplicación es que el sender transfiera un determinado archivo al receiver (véase Section 3.1). Para esto, seguimos los siguientes pasos:

Sender : Verifica que el archivo a transferir exista, abre y se sigue el siguiente procesamiento:

- Iterativamente se van leyendo chunks de máximo 4096 bytes del archivo que se quiere transferir y se pasa esta data a la capa de transporte (RDT) para que envíe el paquete. Este proceso se repite hasta terminar de leer el archivo.
- Finalmente para marcar que ya no se transmitirán más más chunks del file: la aplicación indica al receiver la intención de cerrar la conexión y ya no enviar más paquetes enviando un mensaje con data vacia, de modo que el receiver al obtener data vacia puede dejar de esperar más mensajes del sender.

Receiver : Abre el archivo binario local donde se escribirá lo recibido. Iterativamente se pedir al Reliable Data Transfer (RDT) que reciba un *chunk* (de máximo el tamaño convenido: 4096 bytes) de contenido del archivo en cuestión y se escribe lo devuelto en el archivo abierto.
El proceso se repite hasta que el RDT devuelva un mensaje sin contenido, que en nuestro protocolo de capa de aplicación, indica que el sender indicó que no tiene intenciones de usar más la conexión.

3.8. Implementación a nivel capa de transporte

La implementación interna de los RDTs que usan tanto los clientes como el servidor dependen casi netamente del protocolo de recuperación de errores que se haya configurado para el RDT, con esto las entidades RDT nos sirven principalmente para presentar una interfaz entendible y estándar a las aplicaciones que la quieran usar:

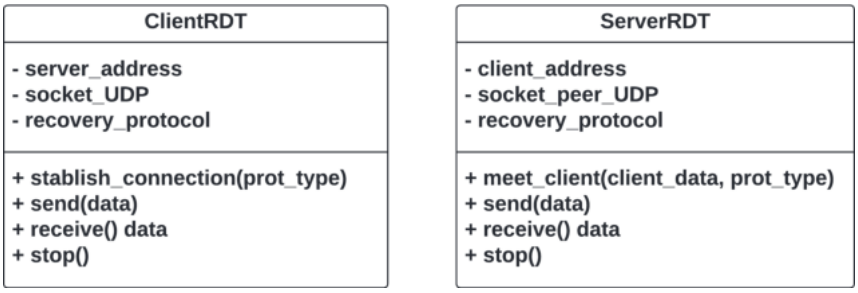


Figura 15: RDT's API

Y a su vez, internamente cada RDT usa alguno de los diferentes protocolos para recuperación de errores (atributo *recovery_error*), sea Stop and wait o selective repeat. Estas dos implementaciones se encargan de ejecutar la transferencia de paquetes de tamaño desconocido para este nivel de abstracción, pues el contenido que las aplicaciones se quieren mandar depende de la lógica y semántica de las aplicaciones y sus mensajes que ya describimos anteriormente, lógica desconocida a este punto.
Es considerando esta última característica mencionada que se decidió imponer en ambas implementaciones internas (de selective repeat y de stop and wait) un tamaño máximo de data que se puede enviar en un segmento (tamaño máximo de payload), pues de trabajar con segmentos muy grandes se podría empeorar la velocidad de transmisión percibida bajo este protocolo debido a que requerirían más datagramas IP para su transmisión, lo cual, además de implicar un procesamiento extra, aumenta la probabilidad de pérdida o corrupción de partes del segmento, lo que obligaría a retransmitir todo el segmento.
El valor en cuestión se determinó en 1024 bytes como máximo para el contenido enviado en un solo segmento udp (además de los header agregados para obtener un reliable data transfer) , buscando alinear este valor con recomendaciones presentes en la documentación oficial, como el RFC 1122, que sugiere tamaños de segmento prudentes para evitar la fragmentación a nivel IP, y el RFC 896, que discute las ventajas de controlar el tamaño de los datagramas para mejorar el rendimiento en redes con cuellos de botella. Además, valores como 1024 bytes son comúnmente utilizados en aplicaciones que no requieren un ajuste dinámico del MSS, permitiendo una implementación simple y eficaz.

Cabe mencionar que estas implementaciones grupalmente se consideron como ubicadas en la frontera entre la capa de aplicación y la capa de transporte, pues, si bien estos protocolos toman responsabilidades concerniente con la capa de transporte, también sucede que está construido sobre un protocolo de transporte existente (UDP) que ataja la parte de multiplexación y demultiplexación.

A continuación se brindará una vista general sobre lo que hacen las implementaciones de recovery_protocol para los métodos receive y send.

Resulta importante mencionar que la diferencia más importante entre el protocolo Selective repeat y Stop and wait es la cantidad de paquetes en vuelo (sin haber recibido su ACK) se permiten enviar desde un sender. Razón por la cual comenzaremos expplicando la lógica de Selective Repeat para tener más de un paquete en vuelo y cómo es que se manejan casos como reordenamiento de paquetes, pérdida de estos y posible corrupción. Finalmente se explicará el porqué Stop and Wait puede ser implementado como una instancia de Selective repeat con tamaño de ventana 1.

3.8.1. Implementación de Selective Repeat

Envio de datos

Por cada paquete a enviar:

- Definimos el ID de paquete y ACK esperado (Base de la ventana).
- Se divide el contenido en una lista de segmentos, agregándole además un segmento vacío que

representará el final del paquete.

- Definimos el conjunto de temprizadores para la espera de ACKs, un conjunto para almacenar los ACKs recibidos en desorden y establecemos el tamaño de la ventana.

Con estos pasos previos, se procede al envío de segmentos

1. El Sender envía tantos segmentos como permita el tamaño de la ventana, siempre que dichos segmentos:
   - No esten a la espera de ACK.
   - No esten ya confirmados (recibidos en desorden)

   Por cada segmento enviado, se establece su respectivo temporizador. Además del payload, cada segmento incluye un encabezado de 8 bytes:
   - SYN: 1b
   - Checksum: 4b
   - Numero de secuencia: 1b
   - ID de paquete: 2b
2. Se recibe el mensaje del cliente, conformado por el ACK y el ID de paquete correspondiente, derivando a los siguientes casos:
   - **ID de paquete recibido es menor al ID de paquete actual:** ACK repetido, lo ignoramos.
   - **ACK recibido es menor al esperado (base de la ventana):** ACK repetido, lo ignoramos.
   - **ACK recibido es mayor al esperado:** Se agrega al set de ACKs recibidos y se elimina del temporizador.
   - **ACK recibido es igual al esperado:** Avanzamos en la ventana y se elimina dicho ack de los temporizadores
3. Verificamos los temporizadores, en caso algun ACK cumpla TIMEOUT, reenviamos el segmento para dicho ACK, actualizandolo en el temporizador.
4. Volvemos al paso 1 hasta confirmar el envío de todos los segmentos del paquete.

Una vez completado el envio de todos los segmentos, avanzamos y continuamos con el envio del siguiente paquete (incrementamos ID de paquete)

Recepción de datos

Por cada paquete a recibir:

- Definimos el ID de paquete y Número de secuencia esperado.
- Definimos un buffer de segmentos para almacenar aquellos recibidos en desorden.
- Definimos el acumulador de datos del paquete a recibir.

Con estos pasos previos, se procede a la recepción de segmentos, derivando en los siguientes casos:

- **Se recibe un segmento con el flag SYN activado:** Se ignora el segmento.
- **Si se recibe un segmento con un ID de paquete anterior al esperado:** Se envía el ACK correspondiente y se ignora su contenido.

- **Se recibe un segmento con numero de secuencia anterior al esperado:** Se envía el ACK correspondiente y se ignora su contenido.
- **Se recibe un segmento con un numero de secuencia mayor al esperado:** Almacenamos su contenido en el buffer de segmentos y enviamos el ACK correspondiente.
- **Se recibe un segmento con el numero de secuencia esperado:** Se agrega al acumulador de datos del paquete y se envía el ACK correspondiente.

  De ser el ultimo segmento del paquete actual, se derivan los datos acumulados a la capa de aplicación y se procede a esperar el siguiente paquete.

  En caso contrario, avanzamos el numero de secuencia segun los segmentos almacenados en el buffer. Si el buffer contiene el último segmento del paquete y el numero de secuencia lo alcanza, entonces realizamos lo mencionado anteriormente y avanzamos de paquete.

3.8.2. Implementación de Stop-and-Wait

Dada la implementación anterior, implementamos Stop-and-Wait como un caso particular de **Selective Repeat, con un tamaño de ventana 1**.

Stop-and-Wait es un caso límite de los protocolos de ventana deslizante. En este esquema:

- Se envía un solo segmento.
- El emisor espera su ACK antes de enviar el siguiente segmento.
- No se permite el envío de múltiples segmentos sin confirmación.

Selective Repeat con ventana de tamaño 1:

- Permite enviar solo un segmento a la vez.
- Si llega su ACK, se avanza la ventana y se puede enviar el siguiente.
- Si no llega el ACK a tiempo, se retransmite ese único segmento.

3.9. Diagrama de clases

Figura 16: Diagrama de clases

4. Pruebas

En la presente sección se describirán las pruebas que se configuraron para testear el reliable data transfer implementado para cliente-servidor, tanto haciendo uso del método de recuperación de errores selective repeat como el método stop and wait. Además se detallarán las configuraciones de cada escenario probado. Las cuales se setearon haciendo uso de la herramienta *mininet* y *miniedit* (proporcionada por *mininet*).

4.1. Casos de prueba: Red pequeña de oficina

Se decidió comenzar probando el sistema en un escenario con una red bastante sencilla: Se simuló una red de oficina pequeña, la cual consta de dos computadoras conectadas por un switch Ethernet sencillo. Este escenario ns permite considerar solo tráfico directo entre los dos hosts, que el delay refleje una latencia minima (dentro de una red local), que la pérdida de paquetes sea casi nula, que se pueda considerar una red estable sin jitter en el delay, que el ancho de banda de todos los enlaces en el camino sean de 100MB . La topología de red se visualizaría como:

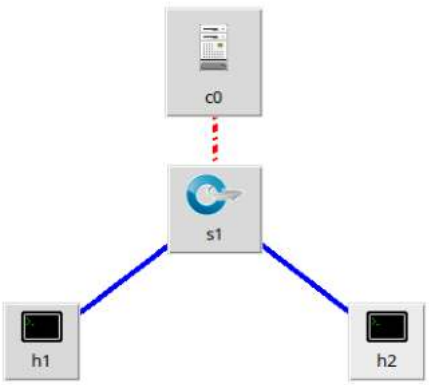
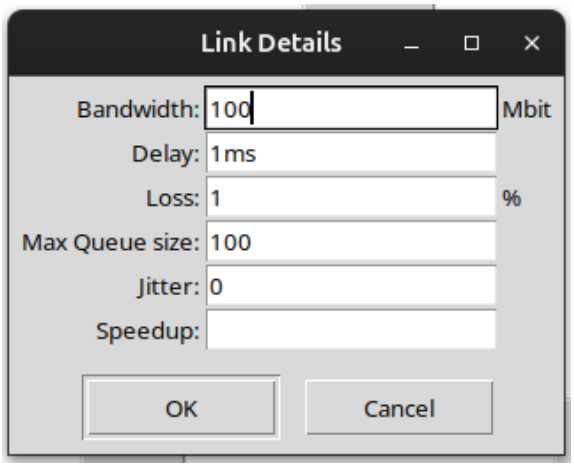


Figura 17: Topología dos hosts un swich

Con la siguiente configuración sobre los enlaces:



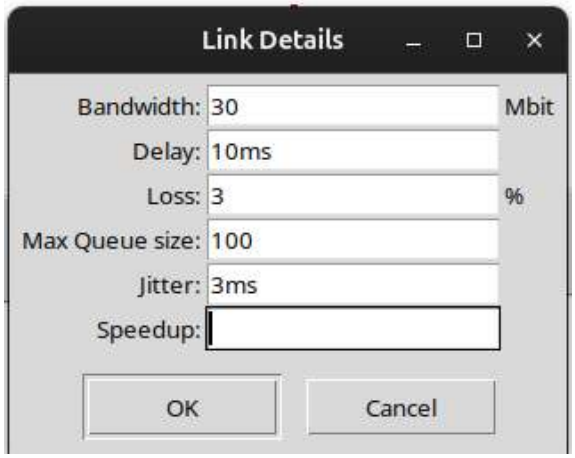
Comportamiento observado del sistema

tamaño de archivo	tiempo con SR	tiempo con SW	dif enviado-recibido SR	dif enviado-recibido SW
1 KB	0.010	0.14	0	0
10 KB	0.027	0.63	0	0
100 KB	0.526	0.584	0	0
1 MB	2.952	5.743	0	0

4.2. Casos de prueba: Oficina Remota Económica

A continuación se prueba el comportamiento del sistema de transferencia bajo el escenario en el que hay dos sucursales conectadas por un ISP local con infraestructura básica, para lo cual tendremos que percibir un tiempo de delay más alto y variable, así como un porcentaje de pérdida más alto.

La topología luce similar a la mostrada superiormente (Section 4.1). Principalmente cambiando los link settings:



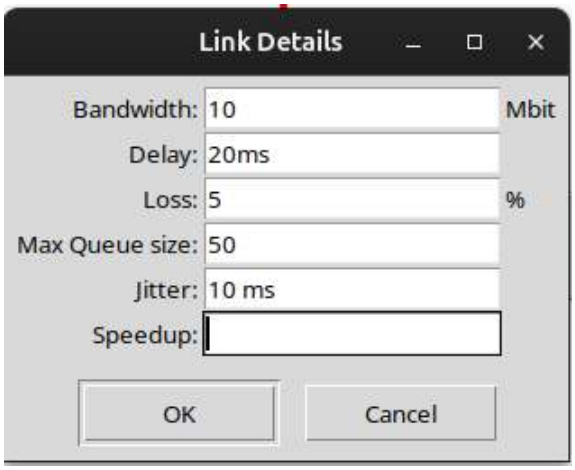
Comportamiento observado del sistema

Editando el timeout a 0.3 segundos.

tamaño de archivo	tiempo con SR	tiempo con SW	dif enviado-recibido SR	dif enviado-recibido SW
1 KB	0.723	0.421	0	0
10 KB	1.026	1.200	0	0
100 KB	8.256	11.697	0	0
1 MB	80.875	95.242	0	0

4.3. Casos de prueba: Red 4G/5G Móvil

Procedemos con la presentación del comportamiento de nuestro sistema bajo el escenario en que la topología tiene características hostiles que ponen a prueba la recuperación de nuestros protocolos. Las características mencionadas son ancho de banda bajo para el medio que enlaza el cliente con el servidor, un delay bastante alto típico en redes móviles además de un jitter notable y una pérdida de paquetes considerablemente alta. Los valores mencionados se concretizan en nuestra simulación en la siguiente configuración sobre todos los enlaces de la topología propuesta previamente (Véase Section 4.1)



Comportamiento observado del sistema

Editando el timeout a 0.5 segundos

tamaño de archivo	tiempo con SR	tiempo con SW	dif enviado-recibido SR	dif enviado-recibido SW
1 KB	0.628	0.619	0	-
10 KB	0.902	1.056	0	0
100 KB	9.066	16.708	0	0
1 MB	94.176	115.871	0	0

Para todos las pruebas con diferentes tamaños la transferencia se realizó íntegramente sin cambios entre lo recibido y enviado.

5. Preguntas a responder

En esta sección responderemos detalladamente las preguntas requeridas para este trabajo.

5.1. Describa la arquitectura Cliente-Servidor

Nuestra aplicación implementa una arquitectura cliente-servidor asimétrica, donde:

- El servidor actúa como nodo central que recibe conexiones de distintos tipos de clientes, mantiene el control de la lógica de negocio y gestiona el flujo de datos.
- Existen dos tipos de clientes con roles diferenciados:
  - El cliente **upload** se conecta al servidor para enviar datos.
  - El cliente **download** se conecta al servidor para recibir datos.

  Esta separación de responsabilidades introduce una forma de comunicación indirecta entre clientes, mediada por el servidor: los datos que sube un cliente upload eventualmente serán consumidos por un cliente download.

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

La función de un protocolo de capa de aplicación es permitir que dos procesos de aplicación, ubicados en extremos distintos de una red, intercambien datos de forma estructurada, comprensible y coordinada, siguiendo un conjunto de reglas estrictamente definidas.

A continuación se detallan las responsabilidades principales de un protocolo de esta capa:

- **Definición de tipos de mensajes**: El protocolo especifica qué clases de mensajes se pueden intercambiar entre las aplicaciones. Estos pueden incluir solicitudes, respuestas, confirmaciones, errores, entre otros.
- **Formato de los mensajes**: El protocolo determina cómo están compuestos los mensajes. Esto incluye la estructura de los *headers* (encabezados), que contienen metadatos esenciales como el tipo de operación, tamaño del mensaje, identificadores, y otros campos necesarios para el procesamiento. También puede incluir un cuerpo (o carga útil) que contiene los datos principales a transmitir.
- **Semántica de los mensajes**: Se establece el significado de cada tipo de mensaje y la acción que debe tomar una aplicación al recibirlo.

- **Reglas de interacción**: El protocolo define cómo debe desarrollarse la comunicación entre los procesos. Esto abarca el orden en que se envían y reciben los mensajes, cómo se maneja el inicio y el fin de la comunicación, los tiempos de espera, la gestión de errores, y posibles retransmisiones o confirmaciones.
- **Interoperabilidad entre aplicaciones heterogéneas**: Al seguir el mismo protocolo, aplicaciones distintas (que pueden estar escritas en distintos lenguajes o ejecutándose en diferentes sistemas) pueden comunicarse correctamente a través de la red.

De esta forma, un protocolo de capa de aplicación permite que las aplicaciones distribuidas en una red cooperen de manera efectiva, definiendo el *qué*, *cómo* y *cuándo* de la comunicación.

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

El protocolo de la capa de aplicación del presente trabajo práctico comprende de lo explicado sobre la arquitectura de software y lo explicado sobre la transferencia de archivos a nivel capa de aplicación.

5.3.1. Tipos de mensajes

El protocolo de capa de aplicación define tres tipos de mensajes:

1. Mensaje de negociación inicial:
   - Enviado por el cliente al inicio de la conexión.
   - Contiene:
     - Tipo de operación: UPLOAD (subir archivo al servidor) o DOWNLOAD (descargar archivo del servidor).
     - Nombre del archivo: Identificador del archivo en el servidor (ej: "documento.txt").
2. Mensajes de datos:
   - Enviado por el programa *sender* (véase qué programa es sender en qué circunstancia)
   - Contiene: fragmentos del archivo en bloques de tamaño *FILE CHUNK SIZE* o por otro lado un mensaje vacío que indica el término de la transferencia.
3. Mensajes de error implícitos:
   - No hay mensajes explicitos para comunicar errores con la finalidad de evitar saturar la red.
   - manejo de errores: por ejemplo, si el archivo no se ha encontrado en se manejan cerrando la conexión o lanzando excepciones.

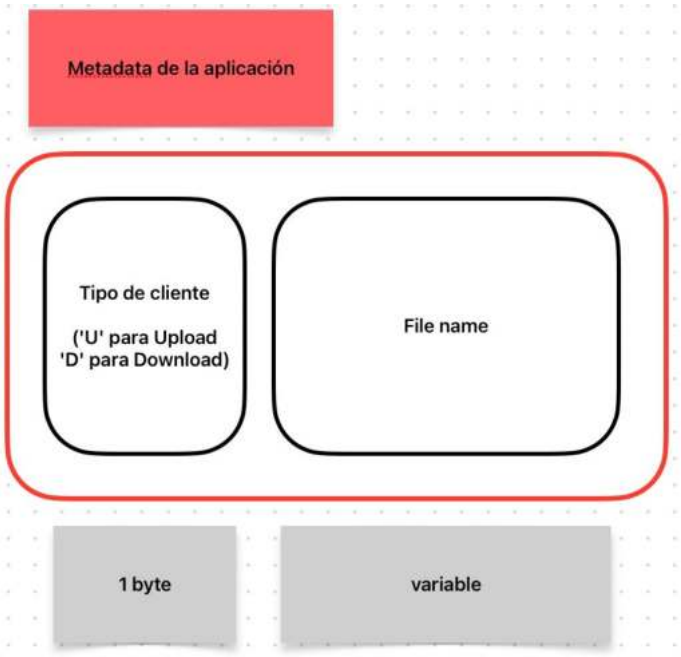


Figura 18: Estructura de la metadata de la app: negociación inicial

5.3.2. Semántica de los Mensajes

- **Mensaje inicial**
  - Cliente → Servidor: Indica la operación a realizar y el archivo objetivo.
  - Servidor: Valida la operación y el archivo (ej: existencia en DOWNLOAD).
- **Mensaje de datos**: El sender envia fragmentos del archivo y el receiver los escribe en disco.
- **Mensaje de datos**: Notifica el fin de la transmisión. Ambos extremos dejan de enviar/recibir datos.

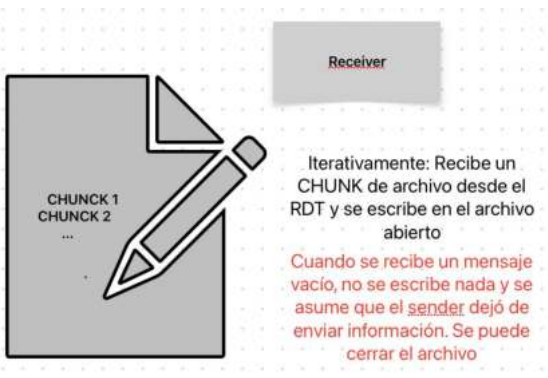


Figura 19: Protocolo de capa de aplicación: receiver

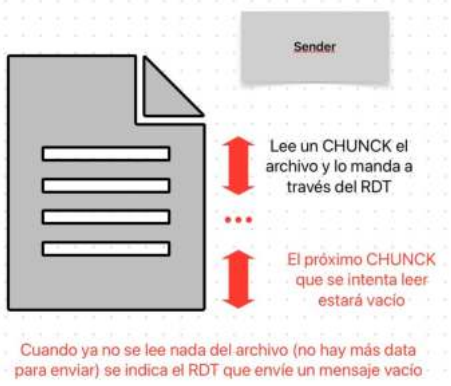


Figura 20: Protocolo de capa de aplicación: sender

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 cda uno?

- **TCP (Transmission Control Protocol)**

Servicios que provee

- **Entrega confiable:** Garantiza que los datos lleguen de manera correcta y completa al destino.
- **Control de flujo:** Regula la cantidad de datos que se pueden enviar antes de recibir una confirmación (evita la sobrecarga de la red).
- **Control de congestión:** Ajusta la velocidad de transmisión según el estado de la red.
- **Orden de los paquetes:** Los paquetes se entregan en el mismo orden en que fueron enviados.
- **Detección y corrección de errores:** Asegura que los datos se reciban sin errores, utilizando sumas de verificación (checksum).
- **Establecimiento de conexión (handshake):** Antes de enviar los datos, TCP establece una conexión entre el emisor y el receptor.

Características

- **Orientado a conexión:** Requiere que se establezca una conexión antes de la transferencia de datos.
- **Confiable:** Asegura que los datos se entreguen correctamente.
- **Lento:** Debido a los mecanismos de control de flujo y corrección de errores, TCP introduce más latencia que UDP.
- **Pesado en recursos:** Requiere mayor uso de memoria y procesamiento para manejar el control de flujo, los ACKs (confirmaciones), retransmisiones, etc.

- **UDP (User Datagram Protocol)**

  Servicios que provee

  - **Entrega no confiable:** No garantiza que los datos lleguen correctamente o en el mismo orden.
  - **Sin conexión:** No establece una conexión previa entre emisor y receptor. Cada datagrama se envía de forma independiente, lo que hace que sea más rápido en cuanto a la transmisión.
  - **No hay control de flujo ni control de congestión:** Los datos se envían sin regulación de la cantidad de información transmitida o el estado de la red.
  - **Sin confirmación de recepción:** No se espera que el receptor confirme la llegada de los datos.

  Características

  - **No orientado a conexión:** No requiere establecer una conexión previa para la transmisión de datos.
  - **Baja sobrecarga y latencia:** Debido a su simplicidad, UDP es más rápido que TCP y usa menos recursos.
  - **Sin garantía de entrega ni orden:** Los datos pueden llegar fuera de orden, perderse, o repetirse sin que el protocolo lo gestione.
  - **No garantiza la integridad de los datos:** Los errores de transmisión no son corregidos de forma automática.

- **Cuándo usar TCP y UDP**

  Cuando utilizar TCP:

  - Aplicaciones que requieren confiabilidad:

- Transferencias de archivos (FTP, SFTP).
- Navegación web (HTTP/HTTPS).
- Correo electrónico (SMTP, IMAP).

- Redes con alta probabilidad de errores o pérdida de datos: Si la red es inestable y se requiere una entrega correcta, es mejor usar TCP debido a su capacidad de manejar retransmisiones y corregir errores.
- Aplicaciones que requieren que los datos lleguen en el mismo orden: Si el orden de los datos es crucial (como en la transmisión de archivos o sesiones interactivas), TCP es más adecuado.

Cuando utilizar UDP:

- Aplicaciones que requieren baja latencia:
  - Juegos en línea.
  - Streaming de video/audio en tiempo real (por ejemplo, VoIP).
- Aplicaciones donde la pérdida de algunos paquetes no es crítica: En muchos casos, la pérdida de algunos paquetes no afecta significativamente el servicio, como en transmisiones de video donde un pequeño retraso es aceptable.
- Aplicaciones que no necesitan una conexión establecida: Si se requiere simplemente enviar datos de manera rápida sin necesidad de que haya confirmación o control de errores, UDP es más eficiente.

6. Anexo

En esta sección, se llevó a cabo la comprobación del proceso de fragmentación de IPv4 mediante la creación de una red virtual.

6.1. Armado de la topología

Utilizando Mininet se armó una topología lineal formada por dos hosts conectados a través de tres switches. Además, se incluyó un controlador para gestionar cómo los switches manejarán el tráfico. Esto se debe a que los switches utilizados son SDN (*Software-Defined Networking*): solo se encargan del envío de datos (plano de datos) una vez que ya saben por dónde deben ir. El controlador centralizado es el que decide qué hacer con los paquetes (por dónde deben ir, qué reglas seguir, etc.), lo cual corresponde al plano de control.

Se hizo uso del editor de redes `miniedit` (proporcionado por Mininet) para facilitar la traducción de la topología de red a un diseño visual:

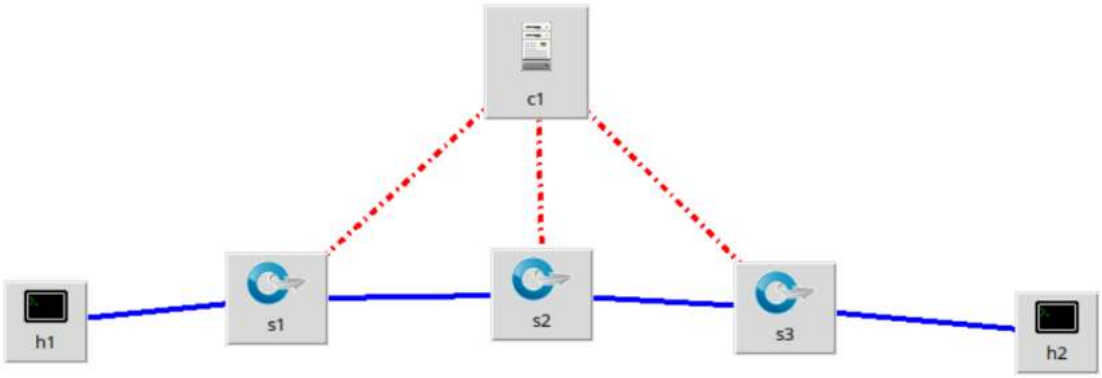


Figura 21: Topología de red de 2 hosts y 3 switches

La topología fue configurada en un script de Python guardado en `/src/topology/linear_ends_topo.py` que se ejecutó de la siguiente manera:

```
sudo mn --mac --custom ./linear_ends_topo.py --topo linends,n,mtu --link tc
```

donde:

`n`: número de clientes (por defecto 1).

`mtu`: MTU (*Maximum Transmission Unit*) para la interfaz (`s2-eth0`) (por defecto 1500).

Al ejecutarlo éste arma la topología y corre la CLI de Mininet:

```
mininet> links
h1-eth0<->s1-eth2 (OK OK)
h2-eth0<->s3-eth2 (OK OK)
s1-eth1<->s2-eth0 (OK OK)
s2-eth1<->s3-eth1 (OK OK)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 s2
h2 -> h1 s2
s2 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Se hizo uso de la IP 10.0.0.1 para el host 1 (`h1`) y la IP 10.0.1.1 para el host 2 (`h2`).

Para visualizar el proceso de fragmentación, se armó la topología variando el valor de MTU de la interfaz `s2-eth0` del switch 2.

A su vez, para comprobar el funcionamiento de UDP/TCP ante la pérdida de un fragmento, se configuró una pérdida de paquetes del 10 % en la interfaz del switch 3 conectado al segundo host (`s3-eth2`):

```
mininet> s3 tc qdisc add dev s3-eth2 root netem loss 10%
```

6.2. Generación de tráfico

Se abrió Wireshark para capturar el tráfico por `s2-eth0`.

Para la generación del tráfico usamos la herramienta *iperf*, que permite medir el rendimiento de la red, generando tráfico TCP o UDP entre dos hosts.

Utilizando *iperf*, se levantó el servidor en el host 1 (10.0.0.1):

```
-- Para UDP
mininet> h1 iperf -s -u

-- Para TCP
mininet> h1 iperf -s
```

donde:

`-s`: levantamos en modo servidor

`-u`: usamos protocolo UDP (por defecto TCP)

Sin fragmentación

Para obtener un tráfico sin fragmentación, se armó la topología dejando el MTU por defecto de 1500. Desde el host 2 (10.0.1.1), que representa al cliente, se mandó un paquete de 1400 bytes (menor al MTU definido, para que el paquete no se fragmente) hacia el servidor:

```
-- Para UDP
mininet> h2 iperf -c 10.0.0.1 -u -l 1400 -t 10

-- Para TCP
mininet> h2 iperf -c 10.0.0.1 -l 1400 -t 0.1
```

donde:

`-c`: a qué dirección IP debe conectarse el host 2 como cliente.

`-l`: en TCP define el tamaño del búfer de envío de cada bloque de datos. En UDP define el tamaño del datagrama. Se mide en bytes.

`-t`: duración de la prueba (en segundos).

La elección de un tiempo de transferencia mayor para UDP que para TCP se debe a que UDP tiene una tasa fija de envío de paquetes (por defecto, 1 Mbps en `iperf` si no se especifica con la opción `-b`), mientras que TCP ajusta la tasa de envío dinámicamente de acuerdo a condiciones de la red como la congestión o la pérdida de paquetes. Esto permite que TCP aproveche mejor el ancho de banda disponible y alcance mayores tasas de transferencia.

Con fragmentación

Para obtener un tráfico con fragmentación, se armó la topología configurando valores de MTU menores a la cantidad de bytes de los paquetes enviados. Se armó la topología con valores de MTU de 500 y de 100. Desde el host 2 (10.0.1.1), que representa al cliente, se mandó un paquete de 1400 bytes.

6.3. Capturas de Wireshark

En Wireshark se visualizaron paquetes enviados de la siguiente manera:

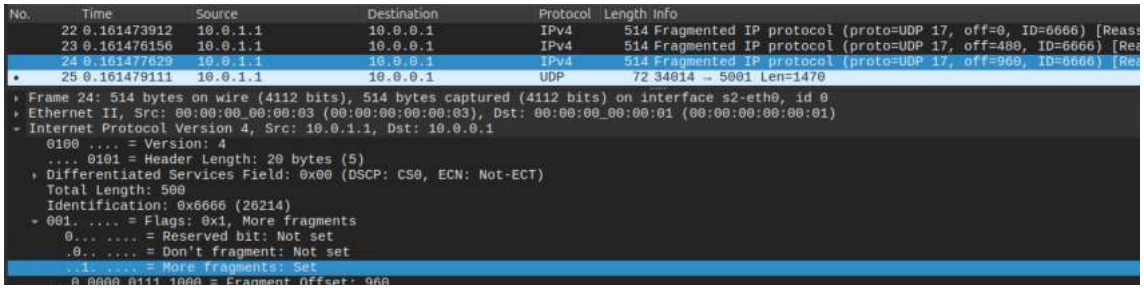
UDP:

No.	Time	Source	Destination	Protocol	Length	Info
882	16.022590828	10.0.1.1	10.0.0.1	UDP	1512	34880 → 5001 Len=1470
883	16.033544126	10.0.1.1	10.0.0.1	UDP	1512	34880 → 5001 Len=1470
884	16.044991994	10.0.1.1	10.0.0.1	UDP	1512	34880 → 5001 Len=1470

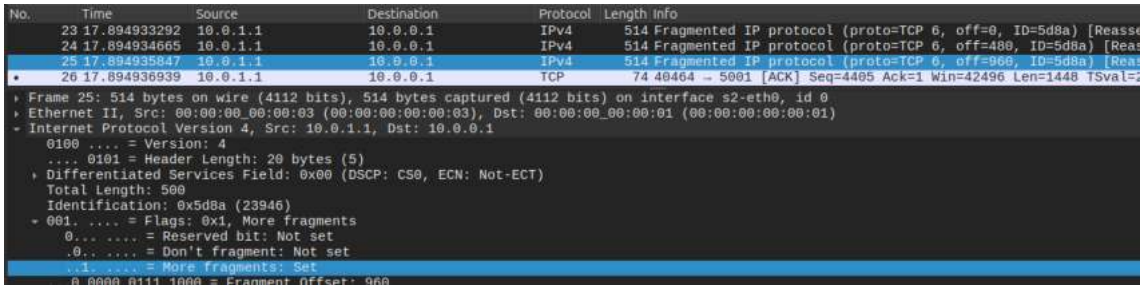
TCP:

No.	Time	Source	Destination	Protocol	Length	Info
638	0.011896903	10.0.0.1	10.0.1.1	TCP	66	5001 → 45816 [ACK] Seq=29 Ack=5741805 Win=7458816 Len=0 TSval=
639	0.011916971	10.0.0.1	10.0.1.1	TCP	66	5001 → 45816 [ACK] Seq=29 Ack=5772213 Win=7458816 Len=0 TSval=
640	0.011922642	10.0.1.1	10.0.0.1	TCP	30474	45816 → 5001 [PSH, ACK] Seq=5772213 Ack=1 Win=42496 Len=30408

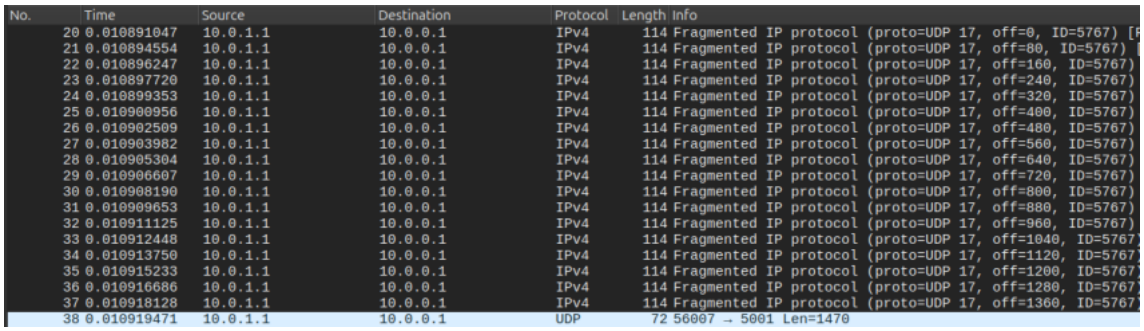
UDP (MTU = 500):



TCP (MTU = 500):

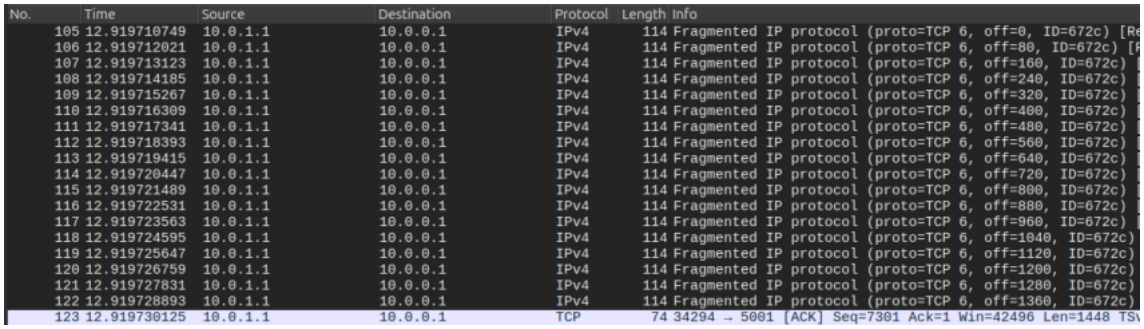


UDP (MTU = 100):



No.	Time	Source	Destination	Protocol	Length	Info
20	0.010891047	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=0, ID=5767) [R
21	0.010894554	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=80, ID=5767) [
22	0.010896247	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=160, ID=5767)
23	0.010897720	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=240, ID=5767)
24	0.010899353	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=320, ID=5767)
25	0.010900956	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=400, ID=5767)
26	0.010902509	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=480, ID=5767)
27	0.010903982	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=560, ID=5767)
28	0.010905304	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=640, ID=5767)
29	0.010906607	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=720, ID=5767)
30	0.010908190	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=800, ID=5767)
31	0.010909653	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=880, ID=5767)
32	0.010911125	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=960, ID=5767)
33	0.010912448	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1040, ID=5767
34	0.010913750	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1120, ID=5767
35	0.010915233	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1200, ID=5767
36	0.010916686	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1280, ID=5767
37	0.010918128	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=UDP 17, off=1360, ID=5767
38	0.010919471	10.0.1.1	10.0.0.1	UDP	72	56007 → 5001 Len=1470

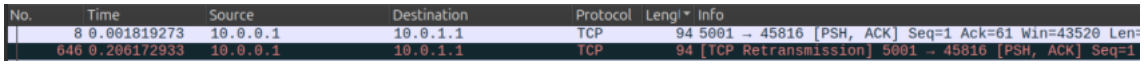
TCP (MTU = 100):



No.	Time	Source	Destination	Protocol	Length	Info
105	12.919710749	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=0, ID=672c) [Re
106	12.919712021	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=80, ID=672c) [R
107	12.919713123	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=160, ID=672c)
108	12.919714185	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=240, ID=672c)
109	12.919715267	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=320, ID=672c)
110	12.919716309	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=400, ID=672c)
111	12.919717341	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=480, ID=672c)
112	12.919718393	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=560, ID=672c)
113	12.919719415	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=640, ID=672c)
114	12.919720447	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=720, ID=672c)
115	12.919721489	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=800, ID=672c)
116	12.919722531	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=880, ID=672c)
117	12.919723563	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=960, ID=672c)
118	12.919724595	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1040, ID=672c)
119	12.919725647	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1120, ID=672c)
120	12.919726759	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1200, ID=672c)
121	12.919727831	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1280, ID=672c)
122	12.919728893	10.0.1.1	10.0.0.1	IPv4	114	Fragmented IP protocol (proto=TCP 6, off=1360, ID=672c)
123	12.919730125	10.0.1.1	10.0.0.1	TCP	74	34294 → 5001 [ACK] Seq=7301 Ack=1 Win=42496 Len=1448 TSv

Con pérdida de paquetes

TCP:



No.	Time	Source	Destination	Protocol	Lengt	Info
8	0.001819273	10.0.0.1	10.0.1.1	TCP	94	5001 → 45816 [PSH, ACK] Seq=1 Ack=61 Win=43520 Len=
646	0.206172933	10.0.0.1	10.0.1.1	TCP	94	[TCP Retransmission] 5001 → 45816 [PSH, ACK] Seq=1

6.4. Análisis

Proceso de fragmentación

Al probar la trasferencia de paquetes sin fragmentación, los paquetes de 1400 bytes que definimos pueden ser enviados completos a través de la red. Esto se puede notar en la columna "Length"de las capturas sin fragmentación de UDP/TCP.

En cambio, al reducir el MTU y enviar paquetes de 1400 bytes, se puede notar la fragmentación.

En el caso de UDP, el protocolo de capa de transporte no fragmenta, sino que entrega el paquete completo a IP (1400 bytes más headers), y es la capa de red quien se encarga de fragmentarlo. TCP sí puede fragmentar los paquetes antes de entregarlos a la capa IP, controlando el MSS (Maximum Segment Size). Pero si el tamaño del segmento supera el MTU de la red, IP también puede fragmentarlo.

Por ejemplo, si se reduce el MTU a 500 e IP detecta que el paquete supera el MTU, lo fragmenta en varias partes, cada una de un máximo de 500 bytes más headers. Cada fragmento lleva el mismo ID, un campo de offset indicando su posición en el paquete original, y un flag "More Fragments"(MF) activado, excepto en el último fragmento.

En los casos que se muestran en la capturas, se tiene un paquete fragmentado en tres partes con `ID` definido y un offset `off` acorde. Este offset se mide en bloques de 8 bytes y solo considera el payload IP, redondeado al múltiplo de 8 más cercano. Por esto el offset en el caso del segmento 23 no resultó igual a 514, sino igual a 480, que es el tamaño del payload ($514 - 20 = 480$).

Todos los fragmentos IP intermedios llevan activada la flag "More Fragments". El paquete 25 corresponde al último fragmento, por lo que tiene esa flag desactivada, indicando el fin de la secuencia. Wireshark muestra en éste último fragmento el paquete UDP/TCP reensamblado, donde definido el campo `Len`, siendo ésta la longitud total del segmento UDP/TCP original.

Funcionamiento de TCP ante la pérdida de un fragmento

La pérdida de un fragmento es detectada por Wireshark si un paquete que ya se había enviado antes con el mismo número de secuencia no fue reconocido (el receptor no envío el ACK en el tiempo esperado).

Ante la pérdida de un paquete, el protocolo TCP hace una retransmisión. Los paquetes que son retransmisiones de marcan con `[TCP Retransmission]`.

Se puede observar en la captura de Wireshark que, luego de enviar el paquete número 8 con número de secuencia `Seq=1`, al no recibir ACK en el tiempo esperado TCP detecto una pérdida, por lo que efectuó una retransmisión del paquete.

Funcionamiento de UDP ante la pérdida de un fragmento

Ante la pérdida de un fragmento, UDP no tiene mecanismos de retransmisión, por lo que si se perdió entonces nunca llegará a destino.

En el caso de tener paquetes fragmentados, si al menos uno de los fragmentos se perdió, entonces el segmento UDP original no puede ser reensamblado y se descarta, por lo que Wireshark no mostrará ningún paquete UDP correspondiente al segmento original.

Aumento de tráfico al reducirse el MTU mínimo de la red

Al reducirse el MTU mínimo de la red (en este caso el MTU del switch 2 se redujo de 500 a 100), se observa un aumento significativo en el tráfico, en la medida en que los paquetes se deberán dividir en más fragmentos para pasar a través de la red.

En las capturas de Wireshark de transferencias mediante protocolos UDP/TCP, se puede notar que, de una partición en 3 fragmentos para un MTU de 500, se pasó a una partición en 18 fragmentos. Esto implica que, por cada segmento UDP/TCP enviado, se transmiten 18 datagramas IP, incrementando la sobrecarga en la red.

7. Conclusión

A partir del desarrollo del presente trabajo práctico, logramos observar que implementar nuestro propio reliable data transfer tiene sus dificultades técnicas como prácticas, el uso de herramientas como *Wireshark* y *Mininet* fueron fundamentales para llevar a a cabo el proceso de testing.

Por otro lado, logramos priorizar las características de confiabilidad que eran fundamentales en para nuestro sistema, es decir, dado que nuestro sistema no requiere de manera crucial de ciertas cualidades de confiabilidad que TCP sí provee, el construir nuestro programa de transferencia de archivos sobre UDP nos permitió dispensar de funcionalidades que nuestro programa no requiere de manera estricta.

Respecto al uso de las herramientas empleadas, tuvimos un control de los bytes que se envíaban y recibían de manera más granular. Como los header de UDP tiene un tamaño fijo y son minimalistas a comparación de los TCP, resultó fácil encontrar los headers agregados por nuestro programa en cada segmento, de forma que pudimos investigar ágilmente qué mensajes se enviaban desde cierta dirección y qué mensajes se recibian del otro lado de la comunicación.

Sobre la capa de aplicación, implementar *Selective Repeat* como protocolo de recuperación de errores maximiza el rendimiento y la robustez de la transferencia de archivos en la mayoría

de los escenarios reales, especialmente cuando el enlace sufre pérdidas moderadas. *Stop-and-Wait* podría servir para implementaciones muy sencillas o entornos extremadamente confiables, pero en términos de throughput y escalabilidad, *Selective Repeat* es la opción óptima.

8. Recursos adicionales

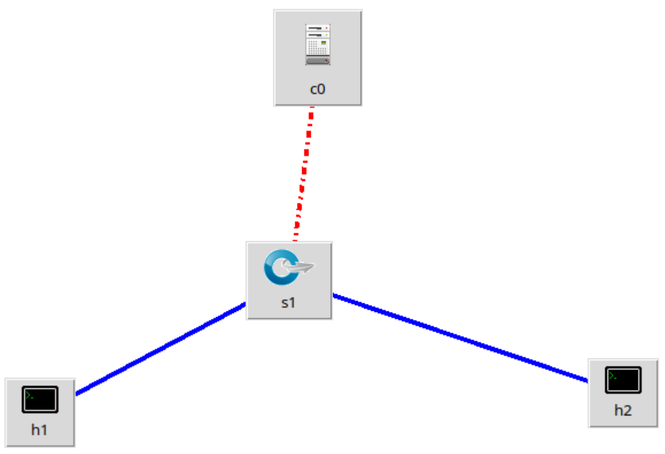
Para más detalles sobre las mediciones obtenidas para las comparativas del SW Y SR, visita el directorio al Drive.

Para más detalles sobre las mediciones obtenidas para el desarollo del anexo (IPv4), visita el directorio al Drive.

Análisis de Protocolos Stop and Wait y Selective Repeat

El siguiente estudio se realiza con las siguientes caracteríasitcas:

- h1: Server siendo receiver
- h2: Cliente siendo sender
- En una topología simple hecha con mininet.
- Con un ancho de banda de:
- TIMEOUT: 0.001
- Tráfico de Wireshark en Enlace s1-eth 0



Contexto desde Wireshark Stop and Wait

El siguiente es un ejemplo del tráfico del envió de un solo paquete.
¿Qué es lo que vemos? Explicación línea por línea.

- Sender envía fragmento.
- Receiver responde con ACK.
- Sender envía fragmento.
- Sender no recibió ACK. Sender envía fragmento de nuevo. [FRAGMENTO PÉRDIDO]
- Receiver responde con ACK.
- Sender envía fragmento.
- Receiver responde con ACK.
- Sender envía fragmento.
- Receiver responde con ACK.
- Sender envía FLAG para indicar termina envío de paquete.
- Receiver responde con ACK.

No.	Time	Source	Destination	Protocol	Length	Info
6612	29.081592…	10.0.0.2	10.0.0.1	UDP	1073	37869 → 51231 Len=1031
6613	29.081662…	10.0.0.1	10.0.0.2	UDP	45	51231 → 37869 Len=3
6614	29.081725…	10.0.0.2	10.0.0.1	UDP	1073	37869 → 51231 Len=1031
6615	29.082895…	10.0.0.2	10.0.0.1	UDP	1073	37869 → 51231 Len=1031
6616	29.082988…	10.0.0.1	10.0.0.2	UDP	45	51231 → 37869 Len=3
6617	29.083070…	10.0.0.2	10.0.0.1	UDP	1073	37869 → 51231 Len=1031
6618	29.083128…	10.0.0.1	10.0.0.2	UDP	45	51231 → 37869 Len=3
6619	29.083197…	10.0.0.2	10.0.0.1	UDP	1073	37869 → 51231 Len=1031
6620	29.083245…	10.0.0.1	10.0.0.2	UDP	45	51231 → 37869 Len=3
6621	29.083324…	10.0.0.2	10.0.0.1	UDP	49	37869 → 51231 Len=7
6622	29.083437…	10.0.0.1	10.0.0.2	UDP	45	51231 → 37869 Len=3

Adicional:
Nuestros fragmentos tienen Len:

- Header: 7
- Data: 1024
  Encabezdos de Protocolos:
  - UDP: 8 bytes (puerto origen + destino + longitud + checksum).
  - IP: 20 bytes (versión, TTL, direcciones IP, etc.).
  - Ethernet (MAC): 14 bytes (direcciones MAC + tipo).

  Total: Length 1073

Selective Repeat

Sender envia fragmento 0
Sender envía fragmento 1
Receiver responde con ACK 0
Receiver responde con ACK 1
Sender envia fragmento 2
Sender envía fragmento 3
Sender no recibió ACK 2 Sender envía fragmento 2 de nuevo [FRAGMENTO 2 PERDIDO]
Sender no recibió ACK 3 Sender envía fragmento 3 de nuevo [FRAGMENTO 3 PERDIDO]
Receiver responde con ACK 2
Receiver responde con ACK 3
Sender envia la FLAG para indicar término de envío del paquete
Receiver responde con ACK

No.	Time	Source	Destination	Protocol	Length	Info
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.2	10.0.0.1	UDP	1…	60278 → 48558 Len=1032
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4
…	0.00…	10.0.0.2	10.0.0.1	UDP	50	60278 → 48558 Len=8
…	0.00…	10.0.0.1	10.0.0.2	UDP	46	48558 → 60278 Len=4

Haz doble clic (o ingresa) para editar

```
!pip install plotly_express
import plotly_express as px
import pandas as pd
import numpy as np

#Visualizaciones
import seaborn as sns
import matplotlib.pyplot as plt

#Modelo Lineal
from sklearn.linear_model import LinearRegression

#Metricas para evaluar modelos
from sklearn import metrics

import pickle
```

```
Requirement already satisfied: plotly_express in /usr/local/lib/python3.11/dist-packages (0.4.1)
Requirement already satisfied: pandas>=0.20.0 in /usr/local/lib/python3.11/dist-packages (from plotly_express) (2.2.2)
Requirement already satisfied: plotly>=4.1.0 in /usr/local/lib/python3.11/dist-packages (from plotly_express) (5.24.1)
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.11/dist-packages (from plotly_express) (0.14.4)
Requirement already satisfied: scipy>=0.18 in /usr/local/lib/python3.11/dist-packages (from plotly_express) (1.15.2)
```

```
Requirement already satisfied: patsy>=0.5 in /usr/local/lib/python3.11/dist-packages (from plotly_express) (1.0.1)
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.11/dist-packages (from plotly_express) (2.0.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.20.0->plotly_express) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.20.0->plotly_express) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.20.0->plotly_express) (2025.2)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.11/dist-packages (from plotly>=4.1.0->plotly_express) (9.1.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from plotly>=4.1.0->plotly_express) (24.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas>=0.20.0->plotly_express) (1.17.0)
```

Fórmula para simplificar métricas

```
def generar_resumen(df, protocolo, mb_archivo, porcentaje_loss):
    # Filtra paquetes de datos (Sender 10.0.0.2)
    data_packets_sender = df[df['Source'] == '10.0.0.2']
    retransmissions = data_packets_sender[data_packets_sender.duplicated(['Length', 'Info'], keep='first')]

    # Cálculo de throughput
    total_data = data_packets_sender['Length'].sum()
    total_time = df['Time'].max() - df['Time'].min()
    throughput = total_data / total_time if total_time > 0 else 0.0

    # Cálculo de RTT (usando el primer intercambio como muestra)
    try:
        data_time = df.iloc[0]['Time']
        ack_time = df.iloc[1]['Time']
        rtt = ack_time - data_time
    except IndexError:
        rtt = None

    # Cálculo de tasa de pérdida
    num_data_packets = len(data_packets_sender)
    num_acks = len(df[df['Source'] == '10.0.0.1'])
    packet_loss = ((num_data_packets - num_acks) / num_data_packets) if num_data_packets > 0 else None

    # Cálculo de tiempo total y fragmentos
    data_packets_large = data_packets_sender[data_packets_sender['Length'] > 1000]
    if not data_packets_large.empty:
        tiempo_total = data_packets_large['Time'].max() - data_packets_large['Time'].min()
    else:
        tiempo_total = 0.0
    total_fragmentos = len(data_packets_large)

    resumen = {
        "Protocolo": protocolo,
        "MB Archivo": mb_archivo,
        "Porcentaje configurado pérdida de paquetes": porcentaje_loss,
        "Tiempo_total_s": round(float(tiempo_total), 3),
        "Total_bytes": int(total_data),
        "RTT_estimado_s": round(float(rtt), 6) if rtt is not None else None,
        "Total_fragmentos": int(total_fragmentos),
        "Bytes_por_segundo": round(float(throughput), 2),
        "Tasa_perdida_%": round(float(packet_loss) * 100, 2) if packet_loss is not None else None,
    }

    return resumen
```

Análisis de Capturas en Wireshark con diferentes tamaños de archivos y pérdida de paquetes

Explicación detallada de cómo se obtienen los datos procesando capturas de Wireshark

Cada .csv es una captura de todo el tráfico que hay entre h1 y h2 con Wreshark al pasarse los archivos detallados con tales pérdidas de paquetes.

- Se explica a detalle cómo se procesan para obtener las métricas.
- Luego usa función que lo hace para no tener código repetido.

Los distintos archivos y pérdida de paquetes con el que probamos protocolos.

- Defino protocolos
- Defino archivos
- Defino porcentajes

```
SELECTIVE_REPEAT="Selective Repeat"
STOP_AND_WAIT="Stop and Wait"


PORCENTAJE_0= "0%"
PORCENTAJE_10= "10%"
PORCENTAJE_20= "20%"


MB_2_6="2.6MB"
MB_5_2="5.2MB"
MB_20_2="20.2MB"
```

Archivo de 2.6 MB

10% Stop and Wait

Pérdida de 10%

```
df_2_6MB_LOSS_10_SW = pd.read_csv('2_6MB_LOSS_10.csv')
```

Inicio y final de todos los fragmentos en Wireshark.

```
df_2_6MB_LOSS_10_SW.head()
```

	No.	Time	Source	Destination	Protocol	Length	Info
0	1	0.000000	10.0.0.2	10.0.0.1	UDP	61	60427 > 12345 Len=19
1	2	0.000713	ea:52:b2:a4:15:39	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
2	3	0.001136	10.0.0.2	10.0.0.1	UDP	61	60427 > 12345 Len=19
3	4	0.001643	fe:4b:a0:16:e1:3b	ea:52:b2:a4:15:39	ARP	42	10.0.0.2 is at fe:4b:a0:16:e1:3b
4	5	0.001657	10.0.0.1	10.0.0.2	UDP	45	50786 > 60427 Len=3

Próximos pasos: Generar código con df_2_6MB_LOSS_10_SW | Ver gráficos recomendados | New interactive sheet

```
df_2_6MB_LOSS_10_SW.tail()
```

	No.	Time	Source	Destination	Protocol	Length	Info
6733	6734	1.314343	10.0.0.1	10.0.0.2	UDP	45	50786 > 60427 Len=3
6734	6735	1.314436	10.0.0.2	10.0.0.1	UDP	49	60427 > 50786 Len=7
6735	6736	1.314515	10.0.0.1	10.0.0.2	UDP	45	50786 > 60427 Len=3
6736	6737	1.314598	10.0.0.2	10.0.0.1	UDP	49	60427 > 50786 Len=7
6737	6738	1.314655	10.0.0.1	10.0.0.2	UDP	45	50786 > 60427 Len=3

Retransmisiones

¿Cómo atrapar los fragmentos retransmitidos?
Si hay envíos duplicados sin ACK en medio. En la foto de ejemplo pasa en los subrayado.

```
df = df_2_6MB_LOSS_10_SW

# Filtra paquetes de datos (Sender 10.0.0.2)
data_packets = df[df['Source'] == '10.0.0.2']
retransmissions = data_packets[data_packets.duplicated(['Length', 'Info'], keep='first')]
print(f"Número de retransmisiones: {len(retransmissions)}")
```

```
Número de retransmisiones: 3456
```

Latencia

Segmentos por segundo

```
total_data = df[df['Source'] == '10.0.0.2']['Length'].sum()  # Bytes enviados
total_time = df['Time'].max() - df['Time'].min()  # Duración total
throughput = total_data / total_time  # Bytes/segundo
print(f"Throughput: {throughput:.2f} B/s")
```

```
Throughput: 2366823.68 B/s
```

Tiempo de respuesta del Receiver

```
data_time = df.iloc[0]['Time']
ack_time = df.iloc[1]['Time']
rtt = ack_time - data_time
print(f"RTT: {rtt:.6f} segundos")
```

```
RTT: 0.000713 segundos
```

Tasa de pérdida

```
num_data_packets = len(df[df['Source'] == '10.0.0.2'])
num_acks = len(df[df['Source'] == '10.0.0.1'])
packet_loss = (num_data_packets - num_acks) / num_data_packets
print(f"Tasa de pérdida estimada: {packet_loss:.2%}")
```

```
Tasa de pérdida estimada: 5.37%
```

Tiene sentido que sea la mitad de pérdida el de un host a otro.

```
df['time_diff'] = df['Time'].diff()
print("Intervalos entre paquetes (s):\n", df['time_diff'].describe())
```

```
Intervalos entre paquetes (s):
 count    6737.000000
mean        0.000195
std         0.000387
min         0.000014
25%         0.000057
50%         0.000073
75%         0.000101
max         0.004878
Name: time_diff, dtype: float64
```

```
# Filtrar paquetes de datos. Así sólo me quedo con los que contienen data.
data_packets = df[(df['Source'] == '10.0.0.2') & (df['Length'] > 1000)]

# tiempo total
tiempo_total = data_packets['Time'].max() - data_packets['Time'].min()

# Contar fragmentos
total_fragmentos = len(data_packets)

print(f"Tiempo total de transferencia: {tiempo_total:.6f} segundos")
print(f"Número total de fragmentos con payload de sender: {total_fragmentos}")
```

```
Tiempo total de transferencia: 1.311193 segundos
Número total de fragmentos con payload de sender: 2873
```

```
protocolo="Stop and Wait"
porcentaje_loss= "10%"

resumen_2_6MB_LOSS_10_SW = {
    "Protocolo": STOP_AND_WAIT,
    "MB Archivo": "2.6 MB",
    "Porcentaje configurado pérdida de paquetes": porcentaje_loss,
    "Tiempo_total_s": round(float(tiempo_total), 3),
    "Total_bytes": int(total_data),
    "RTT_estimado_s": round(float(rtt), 6) if rtt is not None else None,
    "Retransmisiones": int(len(retransmissions)),
    "Total_fragmentos": int(total_fragmentos),
    "Bytes_por_segundo": round(float(throughput), 2),
    "Tasa_perdida_%": round(float(packet_loss) * 100, 2) if packet_loss is not None else None,
}
```

```
resumen_2_6MB_LOSS_10_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '2.6 MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 1.311,
 'Total_bytes': 3111556,
 'RTT_estimado_s': 0.000713,
 'Retransmisiones': 3456,
 'Total_fragmentos': 2873,
 'Bytes_por_segundo': 2366823.68,
 'Tasa_perdida_%': 5.37}
```

```
df_2_6MB_LOSS_10_SW = pd.read_csv('2_6MB_LOSS_10.csv')
```

```
df_2_6MB_LOSS_10_SW= generar_resumen(df_2_6MB_LOSS_10_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_2_6, porcentaje_loss=PORCENTAJE_10)
```

10% Selective Repeat

```
df_2_6MB_LOSS_10_SR = pd.read_csv('2_6MB_LOSS_10_SR.csv')
```

```
r_2_6MB_LOSS_10_SR = generar_resumen(df_2_6MB_LOSS_10_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_2_6, porcentaje_loss=PORCENTAJE_10)
```

```
r_2_6MB_LOSS_10_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 1.274,
 'Total_bytes': 3115719,
 'RTT_estimado_s': 0.000896,
 'Total_fragmentos': 2867,
 'Bytes_por_segundo': 2439047.21,
 'Tasa_perdida_%': 5.61}
```

0% Selective Repeat

```
df_2_6MB_LOSS_0_SR = pd.read_csv('2_6MB_LOSS_0_SR.csv')
```

```
r_2_6MB_LOSS_0_SR = generar_resumen(df_2_6MB_LOSS_0_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_2_6, porcentaje_loss=PORCENTAJE_0)
```

```
r_2_6MB_LOSS_0_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '0%',
 'Tiempo_total_s': 0.319,
 'Total_bytes': 2791390,
 'RTT_estimado_s': 0.000617,
 'Total_fragmentos': 2569,
 'Bytes_por_segundo': 8708082.32,
 'Tasa_perdida_%': 0.03}
```

0% Stop and Wait

```
df_2_6MB_LOSS_0_SW = pd.read_csv('2_6MB_LOSS_0_SW.csv')
```

```
r_2_6MB_LOSS_0_SW = generar_resumen(df_2_6MB_LOSS_0_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_2_6, porcentaje_loss=PORCENTAJE_0)
```

```
r_2_6MB_LOSS_0_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '0%',
 'Tiempo_total_s': 0.408,
 'Total_bytes': 2788177,
 'RTT_estimado_s': 0.000388,
 'Total_fragmentos': 2569,
 'Bytes_por_segundo': 6814705.34,
 'Tasa_perdida_%': 0.03}
```

20% Selective Repeat

```
df_2_6MB_LOSS_20_SR = pd.read_csv('2_6MB_LOSS_20_SR.csv')
```

```
r_2_6MB_LOSS_20_SW = generar_resumen(df_2_6MB_LOSS_20_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_2_6, porcentaje_loss=PORCENTAJE_20)
```

```
r_2_6MB_LOSS_20_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '20%',
 'Tiempo_total_s': 2.326,
```

```
  'Total_bytes': 3394207,
  'RTT_estimado_s': 0.000492,
  'Total_fragmentos': 3140,
  'Bytes_por_segundo': 653647.21,
  'Tasa_perdida_%': 9.62}
```

20% Stop and Wait

```
df_2_6MB_LOSS_20_SW = pd.read_csv('2_6MB_LOSS_20_SW.csv')
```

```
r_2_6MB_LOSS_20_SW = generar_resumen(df_2_6MB_LOSS_20_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_2_6, porcentaje_loss=PORCENTAJE_20)
```

```
r_2_6MB_LOSS_20_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '20%',
 'Tiempo_total_s': 2.326,
 'Total_bytes': 3394207,
 'RTT_estimado_s': 0.000492,
 'Total_fragmentos': 3140,
 'Bytes_por_segundo': 653647.21,
 'Tasa_perdida_%': 9.62}
```

Archivo 5.2 MB

0% Stop and Wait

```
df_5_2MB_LOSS_0_SW = pd.read_csv('5_2MB_LOSS_0_SW.csv')
```

```
r_5_2MB_LOSS_0_SW = generar_resumen(df_5_2MB_LOSS_0_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_5_2, porcentaje_loss=PORCENTAJE_0)
```

```
r_5_2MB_LOSS_0_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '5.2MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 2.607,
 'Total_bytes': 6161172,
 'RTT_estimado_s': 0.000516,
 'Total_fragmentos': 5689,
 'Bytes_por_segundo': 2360633.09,
 'Tasa_perdida_%': 5.04}
```

0% Selective Repeat

```
df_5_2MB_LOSS_0_SR = pd.read_csv('5_2MB_LOSS_0_SR.csv')
```

```
r_5_2MB_LOSS_0_SR = generar_resumen(df_5_2MB_LOSS_0_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_5_2, porcentaje_loss=PORCENTAJE_0)
```

```
r_5_2MB_LOSS_0_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '5.2MB',
 'Porcentaje configurado pérdida de paquetes': '0%',
 'Tiempo_total_s': 0.645,
 'Total_bytes': 5583115,
 'RTT_estimado_s': 0.000447,
 'Total_fragmentos': 5138,
 'Bytes_por_segundo': 8624487.98,
 'Tasa_perdida_%': 0.02}
```

10% Stop and Wait

```
df_5_2MB_LOSS_10_SW = pd.read_csv('_5_2MB_LOSS_10_SW.csv')
```

```
r_5_2MB_LOSS_0_SW = generar_resumen(df_5_2MB_LOSS_10_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_5_2, porcentaje_loss=PORCENTAJE_10)
```

```
r_5_2MB_LOSS_0_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '5.2MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 7.035,
 'Total_bytes': 6151539,
 'RTT_estimado_s': 0.000428,
 'Total_fragmentos': 5680,
 'Bytes_por_segundo': 873726.1,
 'Tasa_perdida_%': 5.12}
```

10% Selective Repeat

```
df_5_2MB_LOSS_10_SR = pd.read_csv('_5_2MB_LOSS_10_SR.csv')
```

```
r_5_2MB_LOSS_10_SR = generar_resumen(df_5_2MB_LOSS_10_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_5_2, porcentaje_loss=PORCENTAJE_10)
```

```
r_5_2MB_LOSS_10_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '5.2MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 6.511,
 'Total_bytes': 6177214,
 'RTT_estimado_s': 0.0007,
 'Total_fragmentos': 5686,
 'Bytes_por_segundo': 948406.05,
 'Tasa_perdida_%': 5.25}
```

20% Stop and Wait

```
df_5_2MB_LOSS_20_SW = pd.read_csv('5_2MB_LOSS_20_SW.csv')
```

```
r_5_2MB_LOSS_20_SW = generar_resumen(df_5_2MB_LOSS_20_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_5_2, porcentaje_loss=PORCENTAJE_20)
```

```
r_5_2MB_LOSS_20_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '5.2MB',
 'Porcentaje configurado pérdida de paquetes': '20%',
 'Tiempo_total_s': 4.643,
 'Total_bytes': 6810889,
 'RTT_estimado_s': 0.000559,
 'Total_fragmentos': 6299,
 'Bytes_por_segundo': 1466190.14,
 'Tasa_perdida_%': 9.81}
```

20% Selective Repeat

```
df_5_2MB_LOSS_20_SR = pd.read_csv('5_2MB_LOSS_20_SR.csv')
```

```
r_5_2MB_LOSS_20_SR = generar_resumen(df_5_2MB_LOSS_20_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_5_2, porcentaje_loss=PORCENTAJE_20)
```

```
r_5_2MB_LOSS_20_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '5.2MB',
 'Porcentaje configurado pérdida de paquetes': '20%',
 'Tiempo_total_s': 4.557,
 'Total_bytes': 6916495,
 'RTT_estimado_s': 0.000658,
 'Total_fragmentos': 6364,
 'Bytes_por_segundo': 827590.94,
 'Tasa_perdida_%': 10.34}
```

Archivo de 20_2MB

Es un vídeo MP4.

0% Stop and Wait

```
df_20_2MB_LOSS_0_SW = pd.read_csv('20_2MB_LOSS_0_SW.csv')
```

```
r_20_2MB_LOSS_0_SW = generar_resumen(df_20_2MB_LOSS_0_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_20_2, porcentaje_loss=PORCENTAJE_0)
```

```
r_20_2MB_LOSS_0_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '20.2MB',
 'Porcentaje configurado pérdida de paquetes': '0%',
 'Tiempo_total_s': 3.083,
 'Total_bytes': 21421606,
 'RTT_estimado_s': 0.000108,
 'Total_fragmentos': 19738,
 'Bytes_por_segundo': 6943438.37,
 'Tasa_perdida_%': 0.01}
```

0% Selective Repeat

```
df_20_2MB_LOSS_0_SR = pd.read_csv('20_2MB_LOSS_0_SR.csv')
```

```
r_20_2MB_LOSS_0_SR = generar_resumen(df_20_2MB_LOSS_0_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_20_2, porcentaje_loss=PORCENTAJE_0)
```

```
r_20_2MB_LOSS_0_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '20.2MB',
 'Porcentaje configurado pérdida de paquetes': '0%',
 'Tiempo_total_s': 2.461,
 'Total_bytes': 21464687,
 'RTT_estimado_s': 0.000653,
 'Total_fragmentos': 19755,
 'Bytes_por_segundo': 8713479.95,
 'Tasa_perdida_%': 0.0}
```

10% Stop and Wait

```
df_20_2MB_LOSS_10_SW = pd.read_csv('20_2MB_LOSS_10_SW.csv')
```

```
r_20_2MB_LOSS_10_SW = generar_resumen(df_20_2MB_LOSS_10_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_20_2, porcentaje_loss=PORCENTAJE_10)
```

```
r_20_2MB_LOSS_10_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '20.2MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 10.07,
 'Total_bytes': 23656751,
 'RTT_estimado_s': 0.000514,
 'Total_fragmentos': 21842,
 'Bytes_por_segundo': 1366874.11,
 'Tasa_perdida_%': 4.97}
```

10% Selective Repeat

```
df_20_2MB_LOSS_10_SR = pd.read_csv('20_2MB_LOSS_10_SR.csv')
```

```
r_20_2MB_LOSS_10_SR = generar_resumen(df_20_2MB_LOSS_10_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_20_2, porcentaje_loss=PORCENTAJE_10)
```

```
r_20_2MB_LOSS_10_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '20.2MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 9.421,
 'Total_bytes': 23747663,
 'RTT_estimado_s': 0.000517,
 'Total_fragmentos': 21856,
 'Bytes_por_segundo': 2519612.45,
 'Tasa_perdida_%': 4.83}
```

20% Stop and Wait

```
df_20_2MB_LOSS_20_SW = pd.read_csv('20_2MB_LOSS_20_SW.csv')
```

```
r_20_2MB_LOSS_20_SW = generar_resumen(df_20_2MB_LOSS_20_SW, protocolo=STOP_AND_WAIT, mb_archivo=MB_20_2, porcentaje_loss=PORCENTAJE_20)
```

```
r_20_2MB_LOSS_20_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '20.2MB',
 'Porcentaje configurado pérdida de paquetes': '20%',
 'Tiempo_total_s': 18.408,
 'Total_bytes': 26318168,
 'RTT_estimado_s': 2.8e-05,
 'Total_fragmentos': 24346,
 'Bytes_por_segundo': 1429414.21,
 'Tasa_perdida_%': 9.91}
```

20% Selective Repeat

```
df_20_2MB_LOSS_20_SR = pd.read_csv('20_2MB_LOSS_20_SR.csv')
```

```
r_20_2MB_LOSS_20_SR = generar_resumen(df_20_2MB_LOSS_20_SR, protocolo=SELECTIVE_REPEAT, mb_archivo=MB_20_2, porcentaje_loss=PORCENTAJE_20)
```

```
r_20_2MB_LOSS_20_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '20.2MB',
 'Porcentaje configurado pérdida de paquetes': '20%',
 'Tiempo_total_s': 16.922,
 'Total_bytes': 26457821,
 'RTT_estimado_s': 0.001567,
 'Total_fragmentos': 24350,
 'Bytes_por_segundo': 1563003.44,
 'Tasa_perdida_%': 10.25}
```

Gráficos

Comparación de Tiempo

Comparación tiempo de los portocolos

```
df_2_6MB_LOSS_10_SW
```

```
{'Protocolo': 'Stop and Wait',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 1.311,
 'Total_bytes': 3111556,
 'RTT_estimado_s': 0.000713,
 'Total_fragmentos': 2873,
 'Bytes_por_segundo': 2366823.68,
 'Tasa_perdida_%': 5.37}
```

```
r_2_6MB_LOSS_10_SR
```

```
{'Protocolo': 'Selective Repeat',
 'MB Archivo': '2.6MB',
 'Porcentaje configurado pérdida de paquetes': '10%',
 'Tiempo_total_s': 1.274,
 'Total_bytes': 3115719,
 'RTT_estimado_s': 0.000896,
 'Total_fragmentos': 2867,
 'Bytes_por_segundo': 2439047.21,
 'Tasa_perdida_%': 5.61}
```

Creación de CSV con los resultados

```
import pandas as pd

# Lista de todos los resultados obtenido con Wireshark
resultados = [
    # 2.6MB
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '2.6MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 1.311, 'Total_bytes': 3111556, 'RTT_estimado_s': 0.000713, 'Retransmisiones': 3456, 'Total_fragmentos': 2873, 'Bytes_por_segundo': 23
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '2.6MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 1.274, 'Total_bytes': 3115719, 'RTT_estimado_s': 0.000896, 'Retransmisiones': 3593, 'Total_fragmentos': 2867, 'Bytes_por_segundo':
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '2.6MB', 'Porcentaje configurado pérdida de paquetes': '0%', 'Tiempo_total_s': 0.319, 'Total_bytes': 2791390, 'RTT_estimado_s': 0.000617, 'Retransmisiones': None, 'Total_fragmentos': 2569, 'Bytes_por_segundo':
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '2.6MB', 'Porcentaje configurado pérdida de paquetes': '0%', 'Tiempo_total_s': 0.408, 'Total_bytes': 2788177, 'RTT_estimado_s': 0.000388, 'Retransmisiones': None, 'Total_fragmentos': 2569, 'Bytes_por_segundo': 681
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '2.6MB', 'Porcentaje configurado pérdida de paquetes': '20%', 'Tiempo_total_s': 2.326, 'Total_bytes': 3394207, 'RTT_estimado_s': 0.000492, 'Retransmisiones': None, 'Total_fragmentos': 3140, 'Bytes_por_segundo': 65
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '2.6MB', 'Porcentaje configurado pérdida de paquetes': '20%', 'Tiempo_total_s': 2.326, 'Total_bytes': 3394207, 'RTT_estimado_s': 0.000492, 'Retransmisiones': None, 'Total_fragmentos': 3140, 'Bytes_por_segundo':
```

```
    # 5.2MB
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '5.2MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 2.607, 'Total_bytes': 6161172, 'RTT_estimado_s': 0.000516, 'Retransmisiones': None, 'Total_fragmentos': 5689, 'Bytes_por_segundo': 23(
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '5.2MB', 'Porcentaje configurado pérdida de paquetes': '0%', 'Tiempo_total_s': 0.645, 'Total_bytes': 5583115, 'RTT_estimado_s': 0.000447, 'Retransmisiones': None, 'Total_fragmentos': 5138, 'Bytes_por_segundo': 8

    {'Protocolo': 'Stop and Wait', 'MB Archivo': '5.2MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 7.035, 'Total_bytes': 6151539, 'RTT_estimado_s': 0.000428, 'Retransmisiones': None, 'Total_fragmentos': 5680, 'Bytes_por_segundo': 87
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '5.2MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 6.511, 'Total_bytes': 6177214, 'RTT_estimado_s': 0.0007, 'Retransmisiones': None, 'Total_fragmentos': 5686, 'Bytes_por_segundo': 9(
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '5.2MB', 'Porcentaje configurado pérdida de paquetes': '20%', 'Tiempo_total_s': 4.643, 'Total_bytes': 6810889, 'RTT_estimado_s': 0.000559, 'Retransmisiones': None, 'Total_fragmentos': 6299, 'Bytes_por_segundo': 14(
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '5.2MB', 'Porcentaje configurado pérdida de paquetes': '20%', 'Tiempo_total_s': 4.557, 'Total_bytes': 6916495, 'RTT_estimado_s': 0.000658, 'Retransmisiones': None, 'Total_fragmentos': 6364, 'Bytes_por_segundo':

    # 20.2MB
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '20.2MB', 'Porcentaje configurado pérdida de paquetes': '0%', 'Tiempo_total_s': 3.083, 'Total_bytes': 21421606, 'RTT_estimado_s': 0.000108, 'Retransmisiones': None, 'Total_fragmentos': 19738, 'Bytes_por_segundo': (
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '20.2MB', 'Porcentaje configurado pérdida de paquetes': '0%', 'Tiempo_total_s': 2.461, 'Total_bytes': 21464687, 'RTT_estimado_s': 0.000653, 'Retransmisiones': None, 'Total_fragmentos': 19755, 'Bytes_por_segundo
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '20.2MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 10.07, 'Total_bytes': 23656751, 'RTT_estimado_s': 0.000514, 'Retransmisiones': None, 'Total_fragmentos': 21842, 'Bytes_por_segundo':
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '20.2MB', 'Porcentaje configurado pérdida de paquetes': '10%', 'Tiempo_total_s': 9.421, 'Total_bytes': 23747663, 'RTT_estimado_s': 0.000517, 'Retransmisiones': None, 'Total_fragmentos': 21856, 'Bytes_por_segundo
    {'Protocolo': 'Stop and Wait', 'MB Archivo': '20.2MB', 'Porcentaje configurado pérdida de paquetes': '20%', 'Tiempo_total_s': 18.408, 'Total_bytes': 26318168, 'RTT_estimado_s': 2.8e-05, 'Retransmisiones': None, 'Total_fragmentos': 24346, 'Bytes_por_segundo':
    {'Protocolo': 'Selective Repeat', 'MB Archivo': '20.2MB', 'Porcentaje configurado pérdida de paquetes': '20%', 'Tiempo_total_s': 16.922, 'Total_bytes': 26457821, 'RTT_estimado_s': 0.001567, 'Retransmisiones': None, 'Total_fragmentos': 24350, 'Bytes_por_segun
]

# Crear DataFrame
df_resultados = pd.DataFrame(resultados)

column_order = [
    'Protocolo', 'MB Archivo', 'Porcentaje configurado pérdida de paquetes',
    'Tiempo_total_s', 'Total_bytes', 'RTT_estimado_s', 'Retransmisiones',
    'Total_fragmentos', 'Bytes_por_segundo', 'Tasa_perdida_%'
]
df_resultados = df_resultados[column_order]

df_resultados.to_csv('resultados_completos.csv', index=False)



df_resultados
```

	Protocolo	MB Archivo	Porcentaje configurado pérdida de paquetes	Tiempo_total_s	Total_bytes	RTT_estimado_s	Retransmisiones	Total_fragmentos	Bytes_por_segundo	Tasa_perdida_%
0	Stop and Wait	2.6MB	10%	1.311	3111556	0.000713	3456.0	2873	2366823.68	5.37
1	Selective Repeat	2.6MB	10%	1.274	3115719	0.000896	3593.0	2867	2439047.21	5.61
2	Selective Repeat	2.6MB	0%	0.319	2791390	0.000617	NaN	2569	8708082.32	0.03
3	Stop and Wait	2.6MB	0%	0.408	2788177	0.000388	NaN	2569	6814705.34	0.03
4	Stop and Wait	2.6MB	20%	2.326	3394207	0.000492	NaN	3140	653647.21	9.62
5	Selective Repeat	2.6MB	20%	2.326	3394207	0.000492	NaN	3140	653647.21	9.62
6	Stop and Wait	5.2MB	10%	2.607	6161172	0.000516	NaN	5689	2360633.09	5.04
7	Selective Repeat	5.2MB	0%	0.645	5583115	0.000447	NaN	5138	8624487.98	0.02
8	Stop and Wait	5.2MB	10%	7.035	6151539	0.000428	NaN	5680	873726.10	5.12
9	Selective Repeat	5.2MB	10%	6.511	6177214	0.000700	NaN	5686	948406.05	5.25
10	Stop and Wait	5.2MB	20%	4.643	6810889	0.000559	NaN	6299	1466190.14	9.81
11	Selective Repeat	5.2MB	20%	4.557	6916495	0.000658	NaN	6364	827590.94	10.34
12	Stop and Wait	20.2MB	0%	3.083	21421606	0.000108	NaN	19738	6943438.37	0.01
13	Selective Repeat	20.2MB	0%	2.461	21464687	0.000653	NaN	19755	8713479.95	0.00
14	Stop and Wait	20.2MB	10%	10.070	23656751	0.000514	NaN	21842	1366874.11	4.97
15	Selective Repeat	20.2MB	10%	9.421	23747663	0.000517	NaN	21856	2519612.45	4.83
16	Stop and Wait	20.2MB	20%	18.408	26318168	0.000028	NaN	24346	1429414.21	9.91
17	Selective Repeat	20.2MB	20%	16.922	26457821	0.001567	NaN	24350	1563003.44	10.25

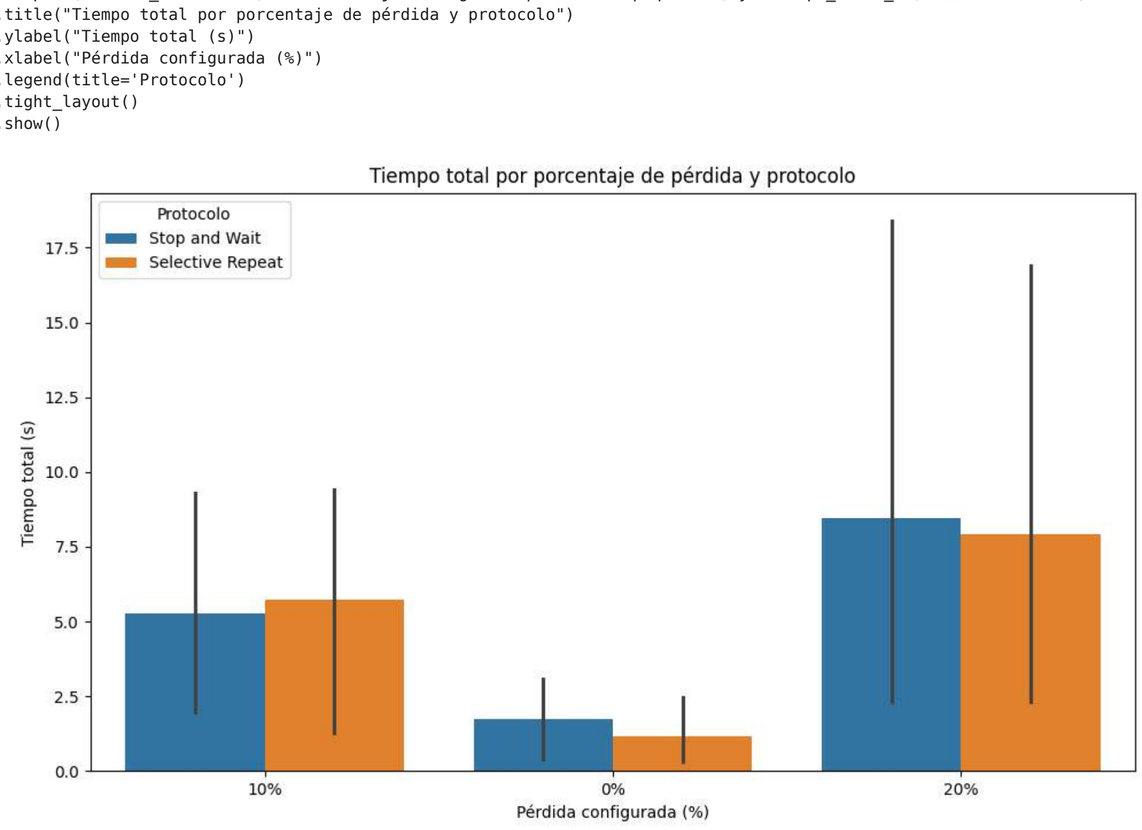
Próximos pasos: Generar código con `df_resultados` | Ver gráficos recomendados | New interactive sheet

Gráficos

Tiempo total por protocolo y pérdida

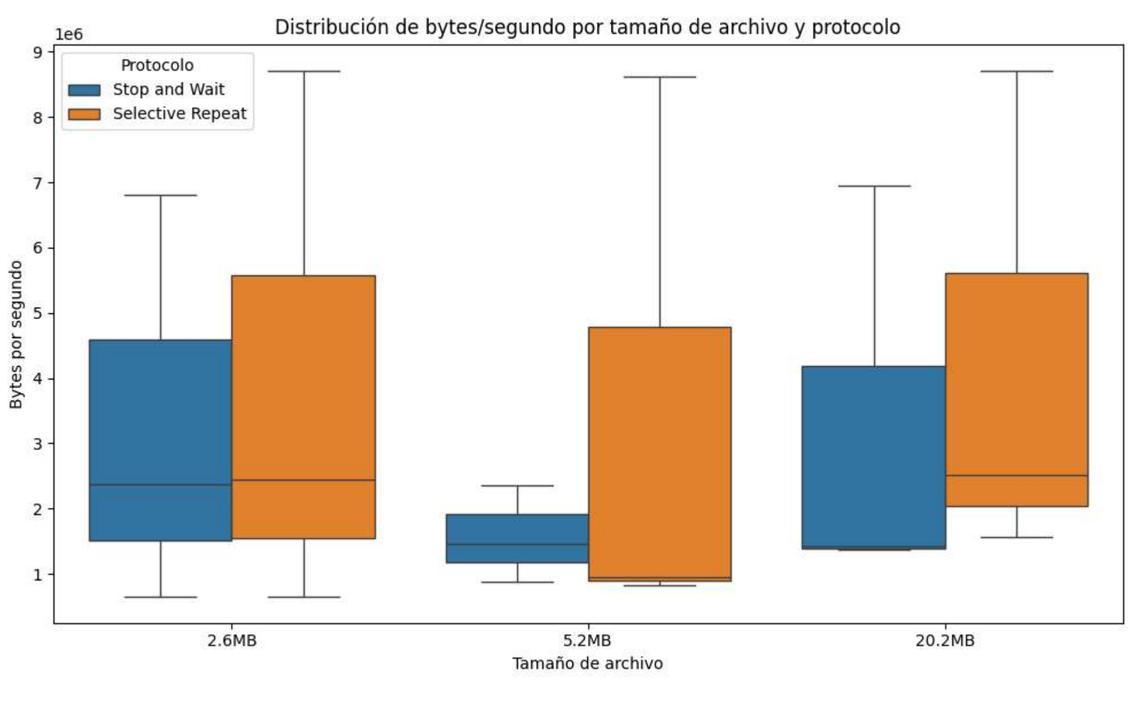
```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10,6))
sns.barplot(data=df_resultados, x='Porcentaje configurado pérdida de paquetes', y='Tiempo_total_s', hue='Protocolo')
plt.title("Tiempo total por porcentaje de pérdida y protocolo")
plt.ylabel("Tiempo total (s)")
plt.xlabel("Pérdida configurada (%)")
plt.legend(title='Protocolo')
plt.tight_layout()
plt.show()
```



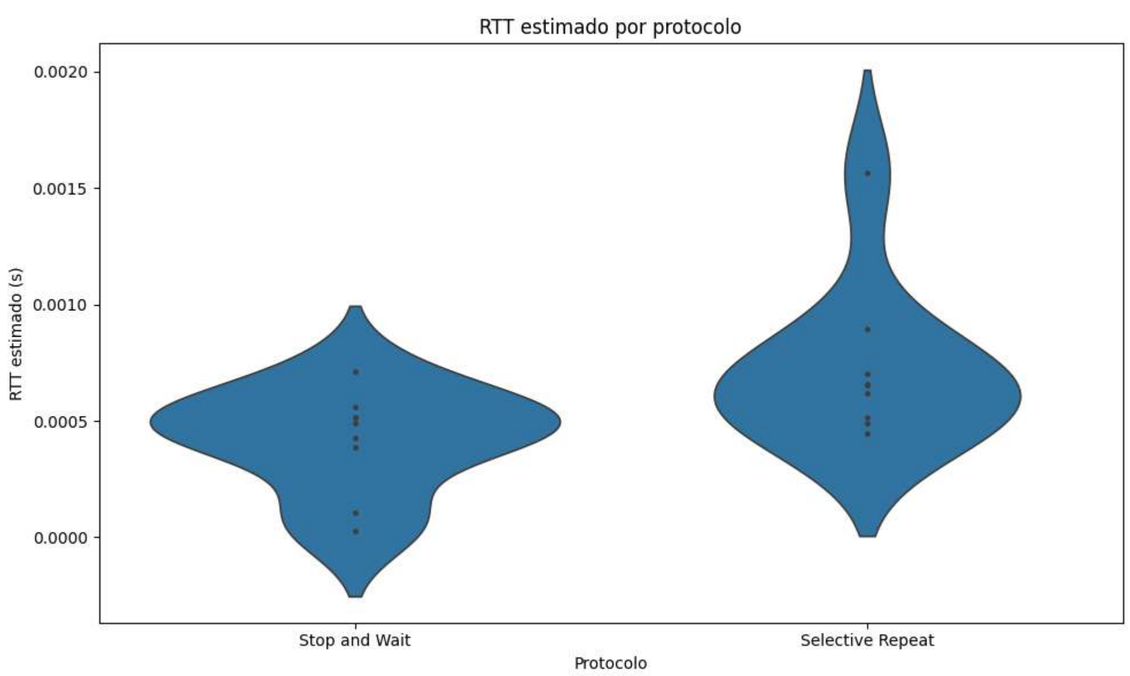
Tiempo de Transferencia según pérdida configurada

```
plt.figure(figsize=(10,6))
sns.boxplot(data=df_resultados, x='MB Archivo', y='Bytes_por_segundo', hue='Protocolo')
plt.title("Distribución de bytes/segundo por tamaño de archivo y protocolo")
plt.ylabel("Bytes por segundo")
plt.xlabel("Tamaño de archivo")
plt.legend(title='Protocolo')
plt.tight_layout()
plt.show()
```



Comparación del RTT estimado por protocolo

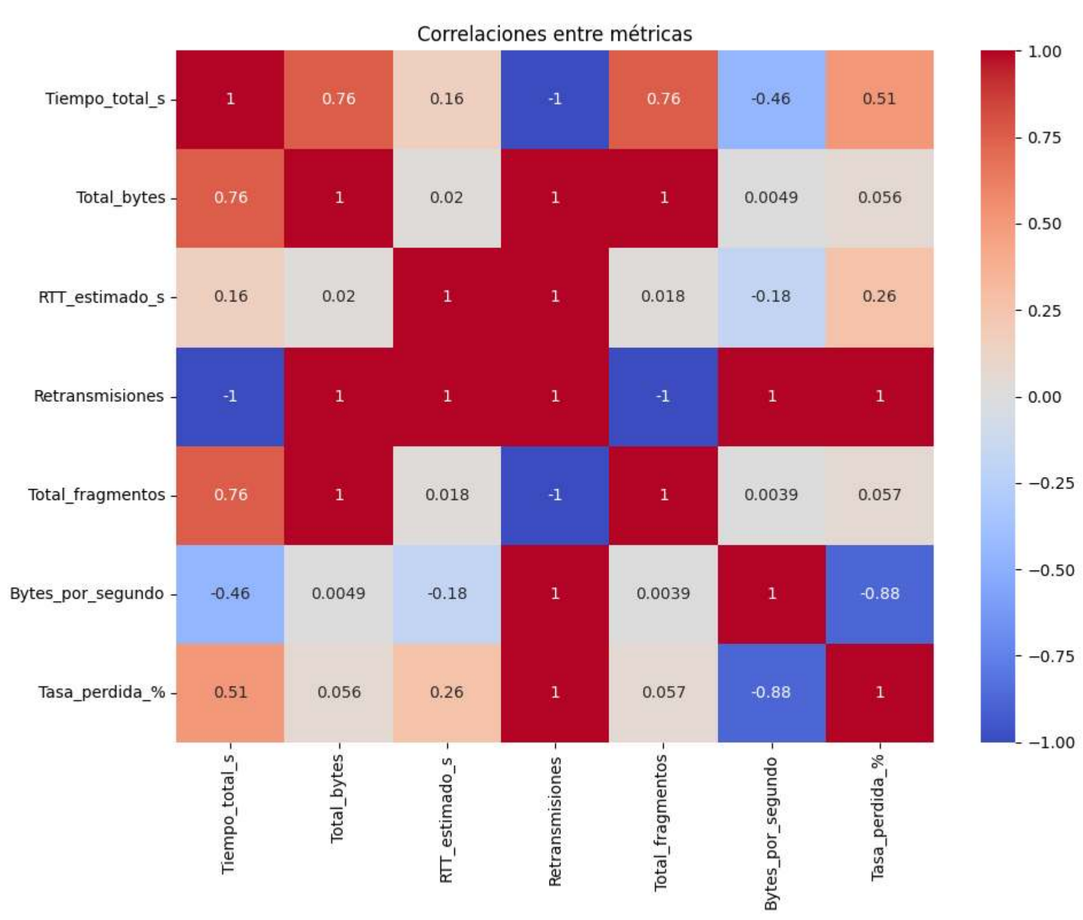
```
plt.figure(figsize=(10,6))
sns.violinplot(data=df_resultados, x='Protocolo', y='RTT_estimado_s', inner='point')
plt.title("RTT estimado por protocolo")
plt.ylabel("RTT estimado (s)")
plt.xlabel("Protocolo")
plt.tight_layout()
plt.show()
```



Correlaciones

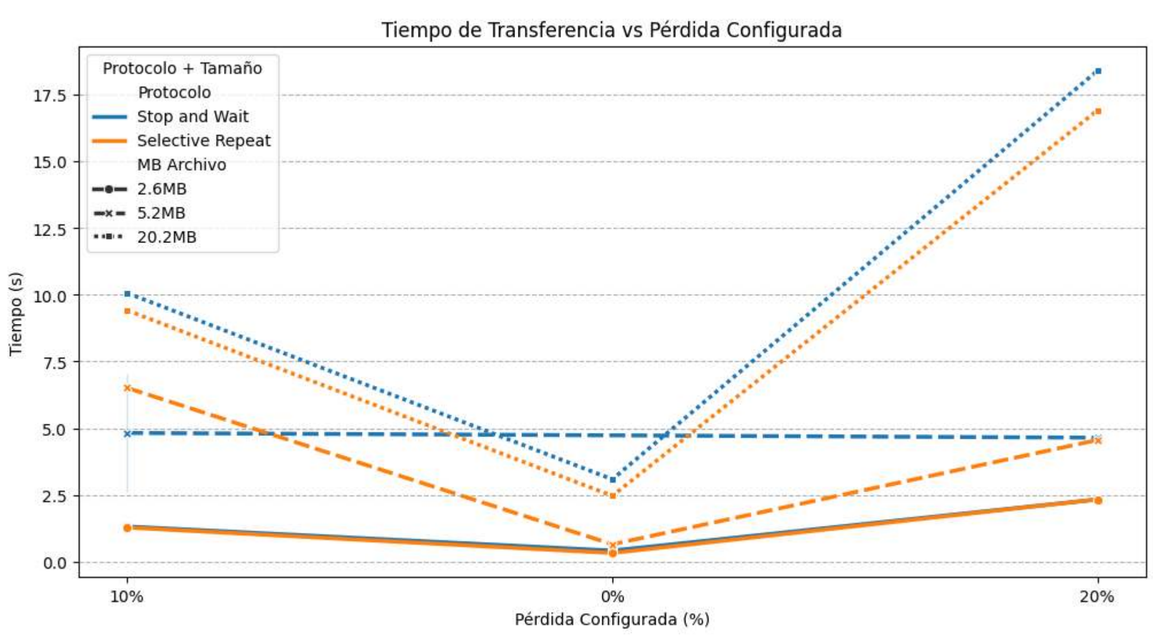
```
import numpy as np

df_numerico = df_resultados.select_dtypes(include=[np.number])
plt.figure(figsize=(10,8))
sns.heatmap(df_numerico.corr(), annot=True, cmap='coolwarm')
plt.title("Correlaciones entre métricas")
plt.tight_layout()
plt.show()
```



Throughput Comparativo (MB/s) por Tamaño de Archivo

```
plt.figure(figsize=(12, 6))
sns.lineplot(
    data=df_resultados,
    x='Porcentaje configurado pérdida de paquetes',
    y='Tiempo_total_s',
    hue='Protocolo',
    style='MB Archivo',
    markers=True,
    palette=['#1f77b4', '#ff7f0e'],
    linewidth=2.5
)
plt.title('Tiempo de Transferencia vs Pérdida Configurada')
plt.ylabel('Tiempo (s)')
plt.xlabel('Pérdida Configurada (%)')
plt.grid(axis='y', linestyle='--')
plt.legend(title='Protocolo + Tamaño')
plt.show()
```

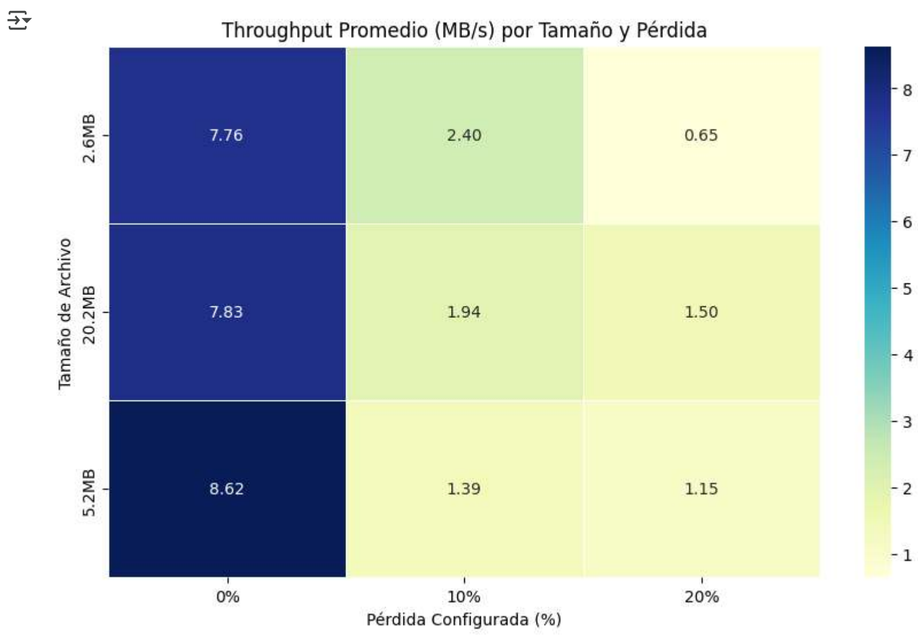


Análisis de périda de paquetes

```
pivot_table = df_resultados.pivot_table(
    index='MB Archivo',
    columns='Porcentaje configurado pérdida de paquetes',
    values='MB_por_segundo',
    aggfunc='mean'
)

plt.figure(figsize=(10, 6))
sns.heatmap(
    pivot_table,
```

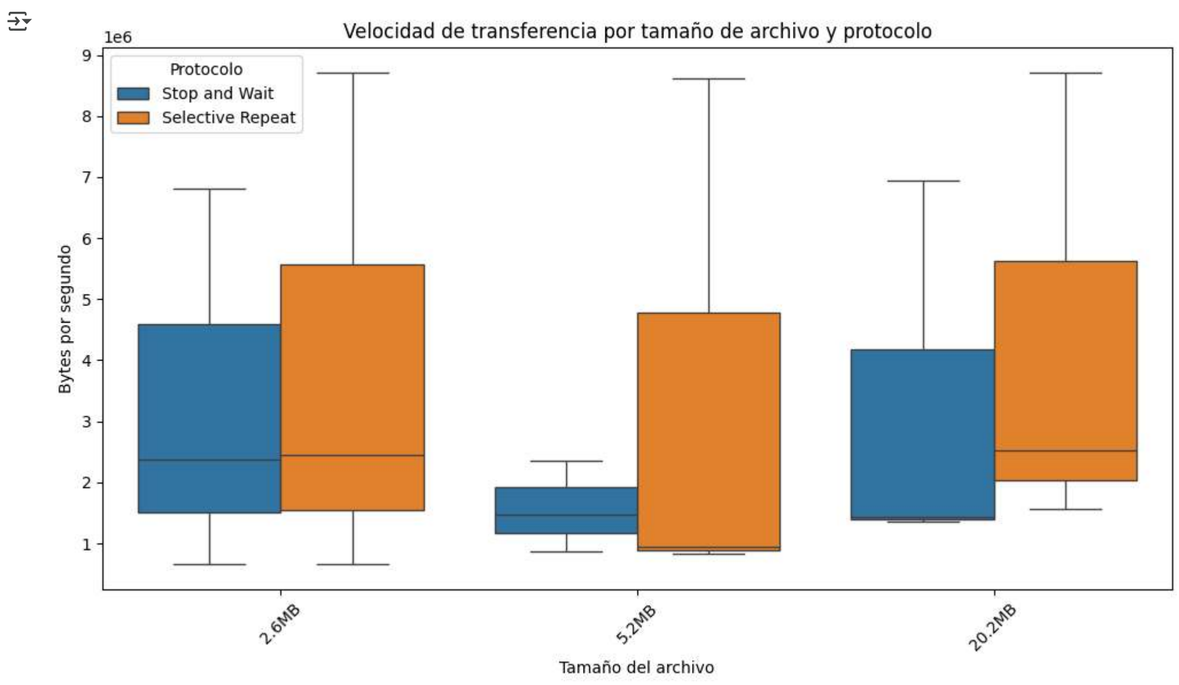
```
    annot=True,
    fmt=".2f",
    cmap="YlGnBu",
    linewidths=0.5
)
plt.title('Throughput Promedio (MB/s) por Tamaño y Pérdida')
plt.xlabel('Pérdida Configurada (%)')
plt.ylabel('Tamaño de Archivo')
plt.show()
```



Las celdas más oscuras (alto throughput) se concentran en pérdidas bajas (0%) y SR.

Pérdidas del 20% impactan más a archivos grandes.

```
plt.figure(figsize=(10, 6))
sns.boxplot(x='MB Archivo', y='Bytes_por_segundo', hue='Protocolo', data=df_resultados)
plt.title('Velocidad de transferencia por tamaño de archivo y protocolo')
plt.ylabel('Bytes por segundo')
plt.xlabel('Tamaño del archivo')
plt.xticks(rotation=45)
plt.legend(title='Protocolo')
plt.tight_layout()
plt.show()
```



Comparación de Tiempos por Protocolo, Tamaño y Pérdida

```
plt.figure(figsize=(14, 7))
ax = sns.barplot(
    data=df_resultados,
    x='MB Archivo',
    y='Tiempo_total_s',
    hue='Porcentaje configurado pérdida de paquetes',
    palette='viridis',
    ci=None,
    estimator='mean'
)
plt.title('Comparación de Tiempos por Protocolo, Tamaño y Pérdida')
plt.ylabel('Tiempo (s)')
plt.xlabel('Tamaño de Archivo')
plt.yscale('log')
plt.grid(axis='y', linestyle='--')
plt.legend(title='Pérdida Configurada', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

```
<ipython-input-438-89d3611ff830>:2: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=None` for the same effect.

  ax = sns.barplot(
```